Department of Computer Science
Series of Publications A
Report A-2021-2

Efficient Approximate String Matching with Synonyms and Taxonomies

Pengfei Xu

Doctoral thesis, to be presented for public examination with the permission of the Faculty of Science of the University of Helsinki, in Auditorium CK112, Exactum, on the 19th of February, 2021 at 14 o'clock.

University of Helsinki
Finland

Supervisor
Jiaheng Lu, University of Helsinki, Finland

Pre-examiners
Wei Wang, University of New South Wales, Australia
Xiaochun Yang, Northeastern University, China

Opponent
Jan Holub, Czech Technical University in Prague, Czech Republic

Custos
Jiaheng Lu, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Pietari Kalmin katu 5)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://cs.helsinki.fi/
Telephone: +358 2941 911

ISSN 1238-8645
ISBN 978-951-51-6987-7 (paperback)
ISBN 978-951-51-6988-4 (PDF)
Helsinki 2021
Unigrafia

Efficient Approximate String Matching with Synonyms and Taxonomies

Pengfei Xu

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
pengfei.xu@helsinki.fi

PhD Thesis, Series of Publications A, Report A-2021-2
Helsinki, January 2021, 62+64 pages
ISSN 1238-8645
ISBN 978-951-51-6987-7 (paperback)
ISBN 978-951-51-6988-4 (PDF)

Abstract

Strings are ubiquitous. When being collected from various sources, strings are often inconsistent, which means that they can have the same or similar meaning expressed in different forms, such as with typographical mistakes. Finding similar strings given such inconsistent datasets has been researched extensively during past years under an umbrella problem called approximate string matching.

This thesis aims to enhance the quality of the approximate string matching by detecting similar strings using their meanings besides typographical errors. Specifically, this thesis focuses on utilising synonyms and taxonomies, since both are commonly available knowledge sources. This research is to use each type of knowledge to address either a selection or join tasks, where the first task aims to find strings similar to a given string, and the second task is to find pairs of strings that are similar. The desired output is either all strings similar to a given extent (i.e., all-match) or the top-k most similar strings.

The first contribution of this thesis is to address the top-k selection problem considering synonyms. Here, we propose algorithms with different optimisation goals: to minimise the space cost, to maximise the selection speed, or to maximise the selection speed under a space constraint. We model the last goal as a variant of an 0/1 knapsack problem and propose an efficient solution based on the branch and bound paradigm.

Next, this thesis solves the top-k join problem considering taxonomy relations. Three algorithms, two based on sorted lists and one based on tries, are proposed, in which we use pre-computations to accelerate list scan or use predictions to eliminate unnecessary trie accesses. Experiments show that the trie-based algorithm has a very fast response time on a vast dataset.

The third contribution of this thesis is to deal with the all-match join problem considering taxonomy relations. To this end, we identify the shortcoming of a standard prefix filtering principle and propose an adaptive filtering algorithm that is tuneable towards the minimised join time. We also design a sampling-based estimation procedure to suggest the best parameter in a short time with high accuracy.

Lastly, this thesis researches the all-match join task by integrating typographical errors, synonyms, and taxonomies simultaneously. Key contributions here include a new unified similarity measure that employs multiple measures, as well as a non-trivial approximation algorithm with a tight theoretical guarantee. We furthermore propose two prefix filtering principles: a fast heuristic and accurate dynamic programming, to strive for the minimised join time.

Computing Reviews (2012) Categories and Subject Descriptors:

Information systems → Data management systems → Information integration → Data cleaning
Information systems → Data management systems → Database management system engines → Database query processing
Information systems → Data management systems → Database design and models → Data model extensions → Inconsistent data

General Terms:
string algorithm, data cleansing, query optimisation, similarity join and selection

Additional Key Words and Phrases:
sampling, estimation, approximate string matching, filtering and verification

Acknowledgements

I would like to express my gratitude and appreciation to my supervisor, Professor Jiaheng Lu, for all the support and guidance through my PhD studies. His conscientious attitude and deep insight into the field has been a vital driving force for me to go towards the PhD degree.

I express my appreciation to the pre-examiners, Professor Wang and Professor Yang, for taking the time to read this manuscript and provide valuable comments to ensure the high quality of this manuscript. I would also like to express my thanks to Professor Holub for becoming the opponent for my PhD defence.

I want to thank all members of the Unified DBMS research group for the valuable discussions that help me develop new ideas and solutions.

The Department of Computer Science, with its friendly staff and cosy atmosphere, has been a joy for me to visit every day. I would like to mention specifically Research Coordinator Pirjo Moen for answering my questions and guiding me through various processes. I also want to thank the Kumpula HR team for the professional support, as well as the IT team for maintaining the (now retired) high-performance cluster Ukko, on which I conducted almost all experiments presented in this manuscript.

I also would like to thank the Department of Computer Science and the Doctoral Programme in Computer Science (DoCS) for providing financial support for the past years. The generous support allows me to concentrate full-time on my research.

Finally, I want to thank my parents, Hengyu Xu and Juan Liu, for all the love that supports me to go through this long journey.

In rainy Amsterdam, January 2021
Pengfei Xu

Contents

List of Publications

This thesis is based on the following original publications, none of which have been included in any other theses. The publications are referred to in texts as Papers I–IV, and have been attached to the end of this thesis.

I. Pengfei Xu and Jiaheng Lu. Top-k string auto-completion with synonyms. In *Database Systems for Advanced Applications - Proceedings of the 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Part II*, pages 202–218, 2017

   The author of this thesis participated in designing the solution, implemented proposed algorithms, conducted experiments, and wrote the paper.

II. Pengfei Xu and Jiaheng Lu. Efficient string similarity join with taxonomy knowledge. *Submitted to Knowledge and Information Systems*, 2019

   The author of this thesis participated in designing the solution, implemented proposed algorithms, conducted experiments, and wrote the paper.

III. Pengfei Xu and Jiaheng Lu. Efficient taxonomic similarity joins with adaptive overlap constraint. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, pages 1563–1566, 2018

   The author of this thesis designed the parameter recommendation algorithm. He implemented proposed methods, conducted experiments, and wrote the paper.

IV. Pengfei Xu and Jiaheng Lu. Towards a unified framework for string similarity joins. *Proceedings of the Very Large Databases Endowment*, 12(11):1289–1302, 2019

The author of this thesis designed the approximation algorithm and the unified filtering framework. He implemented proposed methods, conducted experiments, and participated in writing the paper.

Besides the above papers which contributed to this thesis, the author of this thesis also participated in the following paper:

V. Francesco Concas, Pengfei Xu, Mohammad A Hoque, Jiaheng Lu, and Sasu Tarkoma. Multiple set matching with bloom matrix and bloom vector. *ACM Transactions on Knowledge Discovery from Data*, 14(2):1–21, 2020

The author of this thesis participated in designing the Bloom multifilters and gave theoretical analyses. He conducted experiments, analysed the results, as well as wrote a significant part of the paper.

Chapter 1

Introduction

String, as one of the most common forms of data, has always played a key role in numerous applications. Since being collected from different sources, it is often unpreventable for them to be inconsistent, such as containing typographical errors, synonyms, and related terms. The inconsistencies call for the discovery of similar strings, which becomes a crucial step in many scenarios known as *approximate string matching*. This topic has been attracting researchers' attention for years. Many outcomes have already been applied to many tasks, such as to name a few: auto-completion [12, 21], spell checking [65], duplicate detection [1, 57], and clustering [52]. Increasing use cases illustrate the importance of gaining a deeper insight into the approximate string matching field.

1.1 Motivation

Over the last decades, a variety of contributions are made to solve the approximate string matching problem, mostly focusing on detecting typographical errors. Various similarity measures are proposed, such as *edit similarity* [48], *Jaccard similarity* [11, 24], and *Dice coefficient* [6]. There are also different algorithms that utilise those measures, some of which calculate results directly (e.g., using *prefix/suffix tree* [47]), while others (e.g., [38, 44, 48, 64]) follow the *filtering and verification* procedure which first find potential pairs as candidates and then verify the real similarity of each candidate.

Recently, as the big data era calls for uses of various data collected from abundant sources, string matching considering only typographical errors becomes insufficient forthwith. For example, an entity can have different names in British

and American English or can be described using either specific or general terms. These semantically related terms can also have different spellings, and hence are, unfortunately, ignored by traditional string matching approaches. To enable approximate string matching on these data, recent works [27, 44, 48, 61] are to categorise inconsistencies into three types of similarities:

1. Gram-based similarity [12, 25, 48, 63, 64] measures two strings' literal likeness by splitting each into a set of fixed-length grams and then counts the intersected items between both sets. This type of measure can be used to detect typographic mistakes. As an example, "Helsinki" and "Helsingki" are considered similar by a gram-based measure.

2. Synonym similarity [26,27,58] measures the similarity of strings by leveraging a set of predefined synonym (including synonym, acronym, abbreviation, variation, etc.) rules between strings. For example, "Bill" is a nickname of "William" and "DBMS" is an abbreviation of "Database Management System".

3. Taxonomy similarity [38, 41, 51, 59] measures the similarity of strings by using a *knowledge hierarchy*, an abundant source of IS-A relations. For example, "kitten" is a type of "pet", and "iPhone" is a type of "smartphone". Thanks to the public knowledge bases, e.g., Freebase [43], DBpedia [4], Wikipedia [17], and Yago [42], one can use these available taxonomies to enhance the effectiveness of approximate string matching.

Approximate string matching tasks can be split into two categories: *join* and *selection*. In the first category, the task is to identify all similar string pairs from two lists, where each similar string has a similarity (i) above a predefined threshold (i.e., all-match) or (ii) higher than pairs that are not in the result (i.e., top-k). As the second category, a "selection" task provides a single query string and one string list, while the output strings can be either all-match or top-k strings similar to the query.

Approximate string matching with semantic knowledge is a new research topic, and has not been researched extensively – for example, there is no solution for top-k selection tasks with either synonyms or taxonomies. This situation poses a challenge of filling the remaining vacancies of string matching tasks.

Furthermore, since strings can involve various inconsistencies in real-world applications, considering only one type of similarity can be insufficient to reveal the true similarity between strings. As an example, in Figure 1.1, strings S and

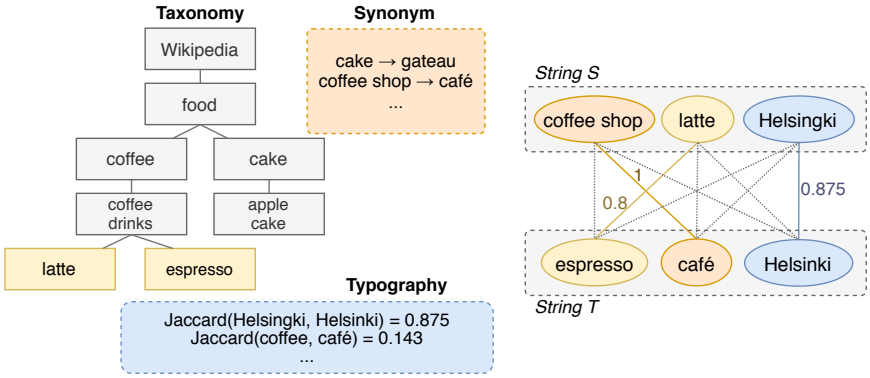


Figure 1.1: Example of strings having synonym, misspelling, and taxonomical relevant terms, simultaneously.

T can be considered to be similar since "coffee shop" is a synonym of "café", both "latte" and "espresso" are coffee drinks, and "Helsingki" is obviously a misspelt "Helsinki". Established methods, in this case, can only surface a portion of all three relations and hence return values much lower than reality. Therefore, this situation calls another challenge of performing string matching tasks by considering multiple similarities altogether.

This thesis aims to address the aforementioned challenges by proposing several string matching algorithms. Specifically, it focuses on answering the following research questions:

RQ1. Given one query string and a string list containing synonymical relevant words, how do we find k string pairs that are the most similar?

RQ2. Given two string lists containing taxonomical relevant words, how do we find k string pairs that are the most similar?

RQ3. Given two string lists containing taxonomical relevant words, how do we find all strings pairs that have similarities greater than a threshold?

RQ4. Given two string lists containing a mixture of typographical, synonymical, and taxonomical relevant words, how do we find all strings pairs that have similarities greater than a threshold?

This thesis contributes several efficient algorithms to solve the above research questions, which are briefly explained in the next section.

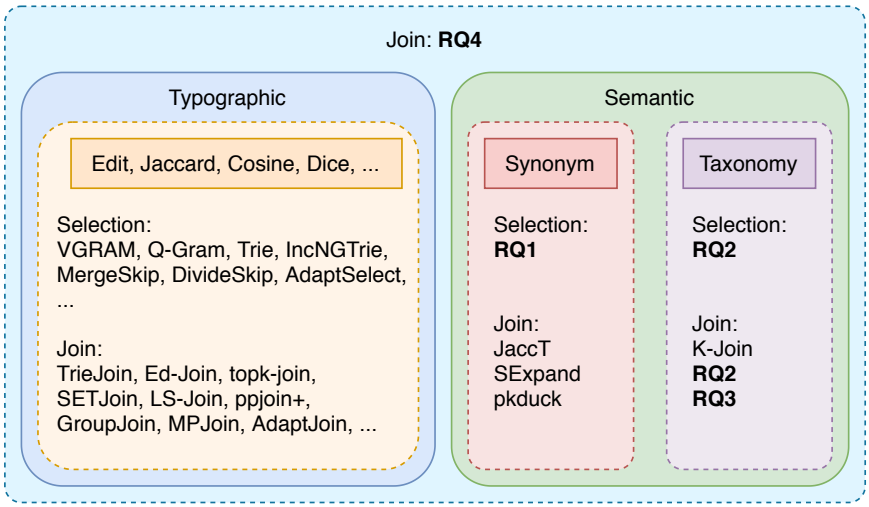


Figure 1.2: Overview of approximate string matching problems and research questions addressed in this thesis.

1.2 Overview

Figure 1.2 shows the established algorithms for solving the approximate string matching problem. For approximate string selection with typographic similarity, numerous pieces of research are established. To name a few: VGRAM [25], which chooses q-grams of variable lengths (based on gram frequencies [25] or cost models [63]) to reduce false positives; Q-Gram and Trie [12], which extend the idea of q-grams and tries to tolerate edit distances for auto-completion tasks. IncNGTrie [53] then improves the performance of the previous method by magnitudes. On the other hand, MergeSkip and DivideSkip [24] utilise lists to allow Jaccard, Cosine, and Dice-based similarities; AdaptSelect [48] uses a time complexity-based cost model to build a proper index. Speaking of the join problem, TrieJoin [47] uses trie as an index to compute similar strings directly; Ed-Join [54] discusses a method of using gram locations to find dissimilar pairs; topk-join [55] and SETJoin [46] aggressively use multiple filtering methods to remove unfeasible pairs; and LS-Join [49] employs a novel local filter [50, 64] that discards string pairs with too many differences. Some other algorithms are also based on the filtering and verification framework, such as ppjoin+ [56, 57], GroupJoin [8], and MPJoin [35]. AdaptJoin [48] stands out from the rest by proposing a variable-length prefix filtering principle to increase the filtering power.

On the semantic side, however, only a little research is conducted. We list some of it here. For synonym-based join tasks: JaccT [1] which enumerate all derived strings to find the maximal similarity, SExpand [26] which appends synonyms to the original string, and pkduck [44] which derives only one string of a pair of strings at a time. For taxonomy-based joins, K-Join [38], which employs the prefix filtering principle, is the only algorithm available to the best of our knowledge.

Figure 1.2 also includes our research problems RQ1–RQ4, which are the main contributions of Papers I–IV. Paper I answers RQ1 by providing several top-k selection algorithms using synonym knowledge. Paper II proposes top-k join algorithms using taxonomies (i.e., RQ2), which are also useful for selection tasks by assuming one input list in the join task contains only the query string. Paper III answers RQ3 by improving the efficiency of all-match join tasks over the state of the art K-Join. Finally, Paper IV is for solving all-match joins by considering multiple similarities simultaneously, defined as RQ4.

The contents of Papers I–IV are summarised as follows.

Paper I: Top-k string auto-completion with synonyms

This paper addresses the problem of finding the top-k strings similar to a given query string. Motivated by a use case of auto-completion, it proposes three methods based on the trie structure to enable the use of synonyms and to reach different aims: fastest response time, least space usage, or as fast as possible given a constrained space. Chapter 3 of this thesis summarises proposed techniques.

Paper II: Efficient string similarity join with taxonomy knowledge

This paper solves the problem of performing top-k join from taxonomy knowledge and can be used for top-k selection tasks. It proposes three new algorithms – two based on sorted lists and one based on tries – to find the top-k similar string pairs efficiently, among which the trie-based algorithm achieves a time complexity linearithmic to the input size. We introduce these techniques in Chapter 4.

Paper III: Efficient taxonomic similarity joins with adaptive overlap constraint

This paper discusses the question of integrating hierarchical taxonomy into the string similarity join process. This paper begins by pointing out a problem of the prefix filtering technique used in the filtering and verification framework, where the lower bound can be tightened up to reduce the number of false positives in candidates dramatically. Based on a tightened bound, this paper adopts the filtering and verification framework to perform joins and confirms

that the efficiency is significantly higher than state of the art. The technical improvements are presented in Chapter 5.

Paper IV: Towards a unified framework for string similarity joins

The last paper proposes a framework for performing typography, synonym, and taxonomy similarities simultaneously in one pass. Furthermore, it introduces a middleware that can be adopted easily for taking in more similarity measures. Experiments on real-world datasets show that the unified framework can find far more similar strings than any single-similarity algorithm. We further discuss the details in Chapter 6.

1.3 Outline

The rest of this thesis is organised as follows. Chapter 2 defines the research problems and introduces some preliminaries. Next, Chapter 3 presents the top-k selection algorithm with synonym similarity, and Chapter 4 for the top-k join/selection algorithm with taxonomy similarity. As for join tasks, Chapter 5 proposes an adaptive algorithm to perform string joins with taxonomy knowledge and furthermore addresses the limitation of a standard prefix filtering strategy. Chapter 6 introduces a unified framework that performs string joins by considering multiple types of similarities, which, as a result, discovers far more similar pairs than any similarity alone. Last, Chapter 7 concludes this thesis and elucidates future work.

Chapter 2

Preliminaries

This chapter gives the formal definition of the research problems solved in this thesis. It also establishes preliminary similarity measures and data structures used in the upcoming chapters.

2.1 Problem Definition

The field of research of this thesis is *approximate string matching*, which can be differentiated into several tasks. First, when the inputs are two string lists and one number as the threshold, we define it as an *all-match join* problem in Problem 2.1.

Problem 2.1 (All-match Approximate String Join) *Given two string lists $\mathcal{S}$ and $\mathcal{T}$, a similarity measure $sim(S,T)$, and a real number $\theta \in [0,1]$. Find a set of string pairs $\mathcal{R}$ consisting of all string pairs $(S,T) \in \mathcal{S} \times \mathcal{T}$ where $sim(S,T) \geq \theta$ holds.*

It is possible to replace the threshold in the input by a positive integer k indicating the maximal result size, thus the problem becomes a *top-k join problem* as defined in Problem 2.2.

Problem 2.2 (Top-k Approximate String Join) *Given two string lists $\mathcal{S}$ and $\mathcal{T}$, a similarity measure $sim(S,T)$, and a positive integer k. Find a set of string pairs $\mathcal{R}$ of at most k items, such that every $(S,T) \in \mathcal{R}$ satisfies $sim(S,T) \geq sim(P,Q)$, where (P,Q) can be any string pair from $\mathcal{S} \times \mathcal{T} \setminus \mathcal{R}$.*

By setting the input list $\mathcal{S}$ to contain only one *query string* S, i.e., $\mathcal{S} = \{S\}$, Problems 2.1 and 2.2 then define the **Approximate String Selection** problems, which are to find either all or top-k strings from $\mathcal{T}$ similar to the query S.

In traditional solutions for both join and selection problems, the similarity measure $\text{sim}(\cdot)$ is often a typographical measure, e.g. edit or Jaccard similarity, and therefore does not handle semantic relations. To this end, the critical challenge is to properly integrate semantic measures into the solution while maintaining the join performance.

2.2 Similarity measures

Let Σ be a universe of *characters*, let $c \in \Sigma$ denote a character, and q be a positive integer. A *string* can be defined as a finite sequence of characters, in the form of $S = \{c_1, c_2, \cdots, c_n\}$. A string can be split into *words*, i.e., subsequences of characters, by continuously splitting each sequence by delimiters such as ' ' (space). The resulting string is in the form of $S = \{s_1, s_2, s_3, \cdots\} = \big\{\{c_1, c_2, c_3\}, \{c_5, c_6\}, \{c_8\}, \cdots\big\}$ where c_4, c_7, and c_9 (and more) are delimiters.

Given two strings split into words, one can utilise a few similarity measures to evaluate their similarity. One straightforward measure is *overlap similarity*, defined as *the number of common words between two strings over the number of distinct words in two strings*. For example, "a bc d" and "b bc e" have an overlap similarity $\frac{1}{5}$ because they have one word "bc" in common among all five distinct words.

The overlap similarity is known to be mutable to inconsistencies [19]. There are several extensions to the measure to tolerate inconsistency, as mentioned below. Note that in the definitions, the term "string" (or S and T) is interchangeable with "word" (or s and t) because a word is also a string.

2.2.1 Gram-based similarity

Gram-based similarity measures quantify the extent that two strings are similar. For example, "Helsingki" spells similar to "Helsinki", whereas "pizza" is not similar to "kitty" at all. To measure such similarity, the gram-based measures work by splitting strings into small chunks called *grams*, so that similar strings must have some grams in common.

Given a string S, its q-grams, denoted by $\text{grams}_q(S)$, is a sequence of words or characters obtained by applying a q-sized sliding window over S. As an example,

the word grams of S when $q = 3$ are $\{s_1, s_2, s_3\}, \{s_2, s_3, s_4\}, \cdots, \{s_{n-2}, s_{n-1}, s_n\}$. Replace S by s and s by c to obtain character grams.

Obtained $\text{grams}_q(S)$ and that of another string T as $\text{grams}_q(T)$, one popular way to measure the similarity between S and T is *Jaccard similarity*, defined as

$$\text{sim}_\text{j}(S, T) = \frac{|\text{grams}_q(S) \cap \text{grams}_q(T)|}{|\text{grams}_q(S) \cup \text{grams}_q(T)|}, \tag{2.1}$$

where $|X|$ returns the number of items in the set X.

Example 2.1 (Word Gram) *Given $q = 1$, the Jaccard similarity between "coffee latte" and "coffee house" is $\frac{1}{3}$, because $grams_1(coffee\ latte) = \{coffee, latte\}$ and $grams_1(coffee\ house) = \{coffee, house\}$ share one word gram "coffee", whereas there are three distinct word grams.*

Example 2.2 (Character Gram) *Given $q = 2$, the Jaccard similarity between "mutt" and "putty" is $\frac{2}{5}$, because $grams_2(mutt) = \{mu, ut, tt\}$ and $grams_2(putty) = \{pu, ut, tt, ty\}$ share two character grams "ut" and "tt", whereas there are five distinct character grams.*

2.2.2 Synonym similarity

A synonym rule R is a one-to-one mapping of two strings in the form of $\text{lhs}(R) \rightarrow \text{rhs}(R)$, $\text{C}(R) \in (0, 1]$, which states that its left-hand side $\text{lhs}(R)$ and right-hand side $\text{rhs}(R)$ has a similarity $\text{C}(R)$. Formally, let $\mathcal{R}$ be a collection of synonym rules, the similarity between two strings S and T is defined as

$$\text{sim}_\text{s}(S, T) = \begin{cases} \text{C}(R), & \text{if } \exists R \in \mathcal{R} \text{ s.t. } \text{lhs}(R) = S \text{ and } \text{rhs}(R) = T; \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

Example 2.3 *Given a synonym rule R_1 : café $\rightarrow$ coffee house, $C(R_1) = 1$, then two strings "café" and "coffee house" becomes equivalent. Furthermore, strings "café in Helsinki" and "coffee house in Helsinki" can been treated as equivalent.*

It is worth mentioning that synonym relations may not be transitive, especially when abbreviations are involved. For example, "DB" can be considered to be synonyms of both "DataBase" and "Deutsche Bahn", which obviously represent different things.

2.2.3 Taxonomy similarity

Given a hierarchical taxonomy, the similarity between two strings can be identified by the positions of matched taxonomy nodes. Let n_S (n_T) be the nodes that matches S (T) and let $|n|$ be the height of node n, the taxonomy similarity between strings S and T is

$$\mathrm{sim_t}(S,T) = \frac{|\mathrm{LCA}(n_S, n_T)|}{\max(|n_S|, |n_T|)}, \tag{2.3}$$

where $\mathrm{LCA}(n_S, n_T)$ returns the *lowest common ancestor*[1] of n_S and n_T.

Example 2.4 *Given two nodes from a geographical taxonomy:* n_1 : *Europe* $\rightarrow$ *Finland* $\rightarrow$ *Helsinki;* n_2 : *Europe* $\rightarrow$ *Finland* $\rightarrow$ *Espoo. Then, the taxonomy similarity between "Helsinki" and "Espoo" is* $\frac{2}{3}$ *because the LCA node of* n_1 *and* n_2 *is "Finland" with height 2, while* $\max(|n_1|, |n_2|) = 3$.

2.3 Index structures and filtering techniques

This section introduces a few popular structures and techniques used for solving string matching problems. They are also used in multiple chapters of this thesis.

2.3.1 Trie

Trie (prefix tree) is a rooted tree structure used for representing multiple strings. The first level of the trie consists of only one "dummy" root; the second level of the trie consists of all of the first characters of strings in the set; the third level of the trie consists of all of the second characters of strings in the set; and so forth. Any non-root node in the trie represents a sequence of characters formed by all its non-root ancestors, starting from the root. To distinguish sequences are strings from the original string list, a common practice is to attach flags to all such nodes. Figure 2.1 depicts a small trie representing four strings.

Trie supports efficient string searches. Take Figure 2.1 as example. String "Vantaa" is not in the list because of no node with the label 'V' in the second level of the trie. Similarly, "Esbo" is also not in the list since there is no node 'b' among the children of *root* $\rightarrow$ E $\rightarrow$ s. In contrast, "Hel" *does* exist in the list since all three nodes are in the trie while the last node is flagged.

[1] `https://en.wikipedia.org/wiki/Lowest_common_ancestor`

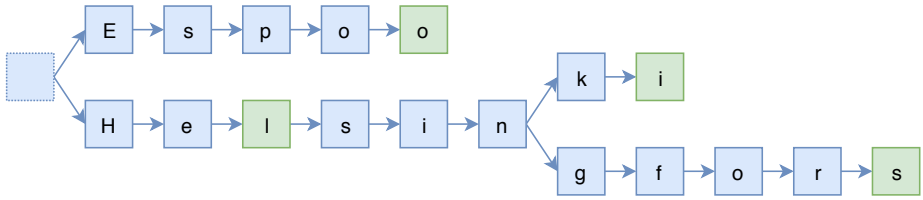


Figure 2.1: A trie representing four strings: "Espoo", "Hel", "Helsinki", and "Helsingfors". The dashed node is the root, green nodes are *flagged*, indicating their sequences are from the original string list.

This thesis extends the standard trie structure in two ways: one to support string search with the consideration of synonym rules (see Chapter 3), and another to enable efficient retrieval of taxonomical relevant strings (Chapter 4).

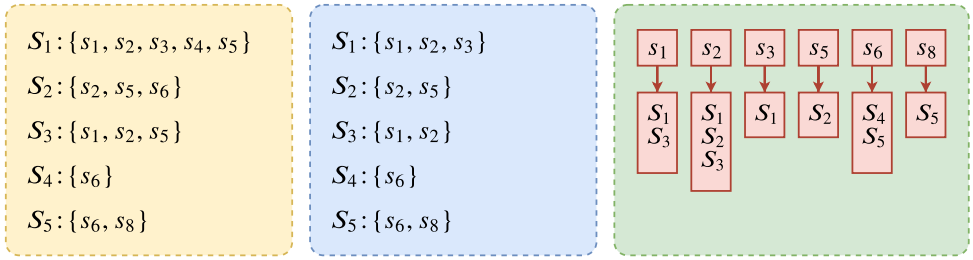
2.3.2 Filtering and verification framework

The filtering and verification framework is a common technique used for solving the approximate string matching problems. Its intuition is that, rather than examining the Cartesian product of all strings, one can identify that some string pairs are "definitely not the answer" thus no need to be examined. Such an identification process is usually fast enough so that the overall time cost becomes lower.

A filtering and verification framework consists of two stages:

1. Filtering: the task of this stage is to identify infeasible pairs that cannot contribute to the answer. Many filtering techniques are available here, to name a few: *prefix filtering* [11], which states that two similar strings must have some common prefix (explained later in this section); *positional filtering* [57], which states that there exists a bond between the difference of positions of the same word in two similar strings; *length filtering* [18], which is based on the fact that the difference of lengths of two similar strings must be within a threshold; and *local filtering* [50, 64], which claims that string with substantial dissimilarities must be dissimilar. This thesis focuses on the prefix filtering since it is proved to be the most effective method for identifying infeasible pairs.

2. Verification: this stage receives string pairs from the filtering stage and calculates the real similarity of each pair. If a pair has a similarity higher than a given threshold, it will be added to the output set.



(a) Input strings in which words are sorted

(b) "Prefix" of each string

(c) Inverted list constructed from prefixes

Figure 2.2: Illustration of the prefix filtering principle where $\theta = 0.6$. Candidates of similar pairs are (S_1, S_2), (S_1, S_3), (S_2, S_3), and (S_4, S_5).

Prefix filtering is one of the most popular techniques used in the filtering stage. It is based on a necessary but not sufficient condition that, after sorting words in strings according to a predefined global ordering, any string similar to a specific string S must share at least one word within the first $(1 - \theta)|S| + 1$ words of S, where θ is the predefined similarity threshold. The following example illustrates the intuition behind this condition.

Example 2.5 *Take five strings in Figure 2.2a as example. Assume that S_2 and another arbitrary S_x have an overlap similarity 0.6. Therefore, according to the prefix filtering principle, S_2 and S_x must have at least one common word: s_2 and/or s_5. This is because otherwise, even when they both have s_6, the similarity is still at most $\frac{1}{3}$ where 3 is the minimal number of distinct words in that case.*

To find those pairs sharing at least one word, one efficient method is by using *inverted lists*, a key-value data structure in which each key is a word, attached by a list of strings containing this word. As an example, after generating prefixes of all strings in Figure 2.2a as Figure 2.2b, it is possible to build one inverted list as in Figure 2.2c. After that, all strings within the same list must have at least one word in common, and hence, according to the prefix filtering principle, can form candidates for further verification.

In Chapters 5 and 6, we extend the prefix filtering technique to handle various similarity measures to solve similarity join problems. Furthermore, we show that prefix filtering can be modified to reduce the number of pairs in the verification stage whilst not damaging the correctness of the output.

Chapter 3

Top-k similarity selection with synonyms

This chapter gives an overview of the results of Paper I that solves RQ1. Following a formal definition of RQ1, it introduces three data structures that solve the problem in different ways with specific optimisation focuses. Finally, this chapter presents empirical experimentation results to highlight the tradeoffs of proposed algorithms.

3.1 Problem formalisation

Paper I formally defines RQ1 as follows: given (i) a set of dictionary strings $\mathcal{S}$ in which each string is assigned a number ("rank") indicating its priority of being selected; (ii) a set of synonym rules $\mathcal{R}$; (iii) a query string S; and (iv) a positive integer k, the task is to output at most k strings from $\mathcal{S}$ satisfying two properties: each string in the output can be obtained from S by applying some synonym rules in $\mathcal{R}$ (see Subsection 2.2.2); and output strings have the top-k highest rank among all possible strings transformable from S.

3.2 Methods

Paper I introduces three trie-based structures to support top-k similarity selection with synonyms. Each structure has a focus: on minimising the running time, on minimising the space cost, or on finding a balance between time and space. We introduce each in turn.

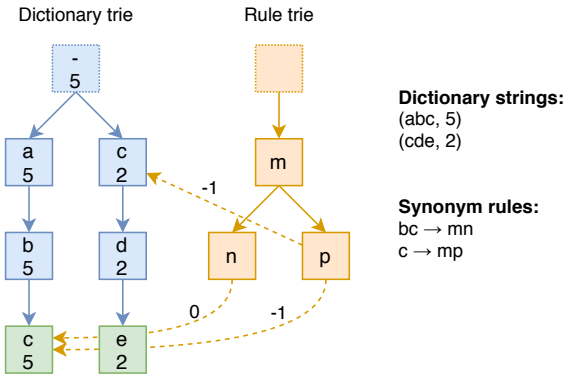


Figure 3.1: A toy Twin Tries (TT) structure representing two dictionary strings, "abc" (rank 5) and "cde" (rank 2); and two synonym rules, "bc → mn" and "c → mp". The number in each node is the rank, dashed arrows are synonym links.

3.2.1 Twin Tries

The Twin Tries (TT) structure aims to minimise the space cost. It consists of two tries: a *dictionary trie* and a *rule trie*, storing all dictionary strings in $\mathcal{S}$ and all right-hand sides (rhs) of synonym rules in $\mathcal{R}$, respectively. On each node in the dictionary trie that represents the last character of a dictionary string, there is a number equal to the rank of that string. From each node in the rule trie that represents the last character of an rhs, there is a synonym link pointing to nodes in the dictionary trie that represent the left-hand side (lhs) of the same rule. Each synonym link is attributed to a number called *delta* which is the length of lhs minus that of rhs. Figure 3.1 shows a toy TT structure.

Section algorithm. The selection procedure presented as Algorithm 2 of Paper I. In short, it continuously looks up the rule trie to find appropriate synonyms, and when an rhs is found, jumps to the corresponding position in the dictionary trie following the synonym link. Its procedure can be summarised as follows:

1. Initialise a heap $\mathcal{P} : \{(n, i)\}$, where n is a trie node and i is an integer. Items in the heap are sorted by rank(n) in descending order;

2. Add $(n_0, 0)$ to $\mathcal{P}$, where n_0 is the root node of the dictionary trie;

3. Pop the head item of $\mathcal{P}$ as (n_p, i). For each $j \in [i, |S|]$, where $|S|$ is the number of characters in the query string S, do the following:

(a) Let n_d be the deepest node below n_p that matches $\{s_i, \cdots, s_j\}$, i.e., from i-th to j-th characters of S;

(b) If n_d is the last character of a dictionary string, add that dictionary string to the result;

(c) If $i = |S|$, i.e., there are no more characters to process in the query string, add each child of n_d to $\mathcal{P}$ as (n_c, i);

(d) Let $\mathcal{N}_r$ be all nodes in the rule trie that in the path of matching the last $|S| - j$ characters of S;

(e) For each $n_r \in \mathcal{N}_r$, and for each synonym link l starting from n_r to a node n_l: if n_d is the same as the node located $\text{depth}(n_r) + \text{delta}(l)$ levels above n_l, add (n_l, j) to $\mathcal{P}$.

4. Go to Step 3 until k results are collected or $\mathcal{P}$ becomes empty.

It can be seen that the final result consists of "exact matches" found in Step 3b as well as "guesses" from Step 3c. All results are arranged by rank in descending order guaranteed by the heap.

3.2.2 Expansion Trie

In the previous TT structure, the rule will be accessed multiple times for finding potential synonym rules. This strategy brings space-saving at the cost of increased running time. In contrast, the Expansion Trie (ET) structure employs a different approach by merging two tries to speed up the selection process: attaching synonym rules to corresponding positions of the dictionary trie. As a result, ET consumes more space but the running time is reduced – as later confirmed by the experimental results in Section 3.3.

Other than TT which uses separate tries for dictionary strings and taxonomy rules, ET mixes them into a single trie, as illustrated in Figure 3.2. The trie is constructed in two passes. In the first pass, all dictionary strings are added to the trie. In the second pass, for each synonym rule and for each dictionary string such a rule can be applied to, it attaches new synonym nodes to the corresponding positions of the trie. For example, given "bc $\rightarrow$ mn" and a string "abc", two new synonym nodes 'm' and 'n' should be attached to 'a', which is the parent node of the first character of rhs. The synonym node 'n' has a synonym link pointing back to 'c', which is the last character of rhs. All synonym nodes have 0 ranks.

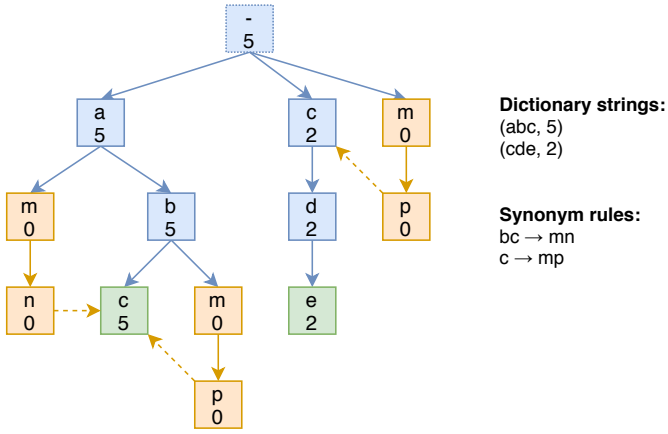


Figure 3.2: A toy Expansion Trie (ET) structure. Nodes with 0 rank come from synonyms.

Search algorithm. The search algorithm on ET is presented as Algorithm 4 of Paper I. The procedure is simpler than that of TT, since ET only has one trie. We present a summarisation of the search algorithm as follows:

1. Initialise a heap $\mathcal{P} : \{n\}$, where n is a trie node. Items in the heap are sorted by rank(n) in descending order;

2. Find n_0 as the deepest node in the trie that matches a prefix of the query string S. Note that this "match" includes synonym nodes but does not follow any synonym link.

3. If n_0 is not a synonym node, add n_0 into $\mathcal{P}$. Otherwise, add to $\mathcal{P}$ the target of synonym links starting from n_0.

4. Pop the head item of $\mathcal{P}$ as n_p.

   (a) If n_p is the last character of a dictionary string, add that dictionary string to the result;

   (b) If n_p is a synonym node, check if depth(n_p) $\leq |S|$, i.e., n_p matches a prefix of the query string. If yes, add all targets of synonym links starting from n_p to $\mathcal{P}$;

   (c) If n_p is a dictionary node and depth(n_p) $< |S|$ (i.e., n_p is not the last character in the query string), search for the next character among the

children of n_p, and (if found) add it to $\mathcal{P}$. If the next character is not found, add all children of n_p to $\mathcal{P}$.

5. Go to Step 4 until k results are collected or $\mathcal{P}$ becomes empty.

The search algorithm of ET also considers both "exact matches" found in Step 4a and "guesses" from the second case of Step 4c.

3.2.3 Hybrid Trie

While TT and ET focus on either space-saving or fast lookup, Paper I proposes the third structure to strike for a balance in between. The idea is to expand some of the synonym rules in the ET favour (to accelerate lookup) while leaving others in a separate rule trie (to save space). The key is to expand rules that appear frequently in dictionary strings to expect more speed benefits.

A Hybrid Trie (HT) asks the user to give a constraint P as the additional space for storing synonym nodes. Then, the rule selection procedure can be modelled as a *0/1 knapsack problem* where (i) P is the capacity of the knapsack; (ii) each synonym rule is an item to be packed; (iii) the frequency of a rule is the item value; and (iv) the space required for expanding a rule is the item weight. However, measuring the weight is not straightforward because some rules are "interfering" with others [9], such as in the case described in Example 3.1.

Example 3.1 *Given a dictionary trie of string "abc" and two rules* $R_1 : bc \to mn$, $R_2 : bc \to mnp$. *Expand either* R_1 *or* R_2 *requires* 4 *or* 5 *units of space, respectively. On the other hand, expanding both* R_1 *and* R_2 *require only* 5 *units of space, instead of* $4 + 5 = 9$, *because* R_1 *and* R_2 *attach to the same position in the dictionary trie and hence share taxonomy nodes 'b', 'c', 'm', and 'n'.*

To calculate item weights correctly regardless of inferences, the rule selection procedure has to be modelled as a variant of the 0/1 knapsack problem, where item weights are determined by a function:

$$\max \sum_{i=1}^{|\mathcal{R}|} v_i x_i \;\text{ subject to }\; \sum_{i=1}^{|\mathcal{R}|} w_i(x_i|x_j, \text{ where } j \in I_i) \leq P$$

where R_i is the i-th synonym rule in $\mathcal{R}$, x_i is a binary indicator for whether R_i is expanded ($x_i = 1$) or not ($x_i = 0$), v_i is the frequency ("value") of R_i in all dictionary strings, and $w_i(\cdot)$ is a function that returns the actual space cost ("weight") to expand R_i given I_i consisting of indices of all rules which are

interfering with R_i. The optimal solution is a series of indicators $\{x_1, \cdots, x_{|\mathcal{R}|}\}$ that maximises the sum of frequencies of all expanded rules within the space limitation P.

Branch and bound algorithm. Paper I proposes a heuristic based on the branch and bound paradigm [23]. Intuitively, given a brute-force full decision tree where each node stands for a rule R_i and has two children indicating $x_i = 0$ and $x_i = 1$, the heuristic traverses the tree and, each time when visiting a node n_i, calculates the value lower and upper bound for each of $x_i = 0$ and $x_i = 1$: $(\mathrm{LB}_i, \mathrm{UB}_i)|x_i = 0$ and $(\mathrm{LB}_i, \mathrm{UB}_i)|x_i = 1$. In either decision regarding x_i, all subsequent decisions can be skipped if $\mathrm{UB}_i < \mathrm{LB}_j$ where $j \in [1, |\mathcal{R}|]$ holds, indicating that the current decision will not lead to the maximal value.

Bound calculation. The value bound is calculated by employing multiple estimation algorithms for knapsack problems. As a prerequisite, let $\mathcal{R}'_{\min}$ ($\mathcal{R}'_{\max}$) be a list of $\{R_i, \cdots, R_{|\mathcal{R}|}\}$ sorted in ascending order by weight and assume that all (none) of the interfering rules for each of them are already expanded. Then:

- The value upper bound UB_i at stage R_i is obtained by adapting the solution for the *fractional knapsack problem* [15]. Specifically, UB_i equals the sum of (i) values of rules in the knapsack from the previous decisions and (ii) the sum of values by greedy taking rules in $\mathcal{R}'_{\min}$ as a whole or as a fraction into the knapsack until the capacity is exhausted.

- To calculate the lower bound LB_i, we adopt the greedy solution of the 0/1 knapsack problem. In this case, LB_i is the sum of (i) values of rules in the knapsack from the previous decisions and (ii) the sum of values by greedily taking rules in $\mathcal{R}'_{\max}$ as a whole into the knapsack until the capacity is exhausted.

Example 3.2 *Figure 3.3 shows the HT structure representing two dictionary strings "abc" (rank 5) and "cde" (rank 2) and two synonym rules "bc → mn" and "c → mp".*

Section algorithm. The selection procedure for HT is very similar to that for TT, with a additional logic that after Step 3b, the algorithm should add the targets n_d of synonym links starting from n_p into the heap as (n_d, i).

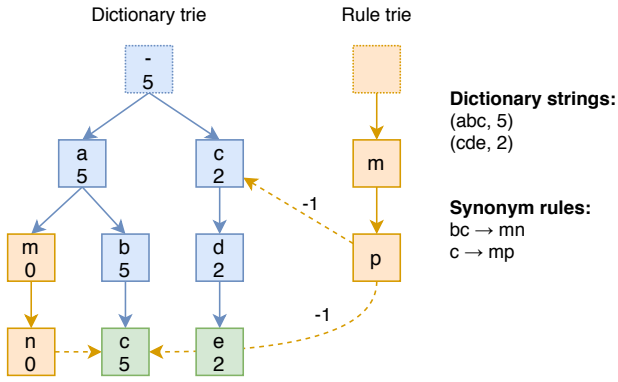


Figure 3.3: A toy Hybrid Trie (HT) structure. Nodes with 0 rank come from synonyms.

3.3 Comparison of methods

This section presents some important experimental results of Paper I to highlight the differences between TT, ET, and HT. We compare three algorithms by structure build time, structure space cost, as well as top-k selection time.

Dataset and setup. The experiments employ three datasets, namely (i) DBLP: 24,810 dictionary strings of book titles and conference names, with 214 synonym rules of common abbreviations; (ii) USPS: 1M artificial addresses consist of person, street, city, and state names, along with 341 synonym rules, e.g. "TX → Texas"; and (iii) SPROT: dataset of 1M gene/protein records, each formed by an entry name, a protein name, a gene name, and an organism name. 1,000 synonyms are used for this dataset. We ran experiments on a Ubuntu machine with Intel i7-4770 3.4GHz processor and 8GB RAM. We set the space threshold of TT to $0.5 \times (S_{ET} - S_{TT})$, i.e., half of the space differences between TT and ET.

Trie build time. The first difference of proposed methods is the speed for constructing the trie structure. As shown in Figure 3.4, all of TT, ET, and HT are faster to build than using BL, a baseline which applies all permutations of rules to each dictionary string and collects transformed strings to form an ET structure. Apart from that, TT is the fastest because it does not do any expansion, but instead builds two tries and adds synonym links. HT, on the other hand, has spent some time to run the branch and bound algorithm to decide which rules are to be expended. All three structures are constructed within one minute given large 1M datasets.

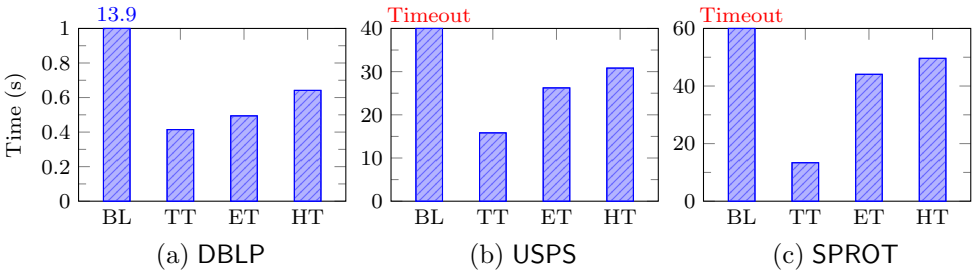


Figure 3.4: Time required for constructing each structure. "BL" is a baseline that generates all possible strings by permutation.

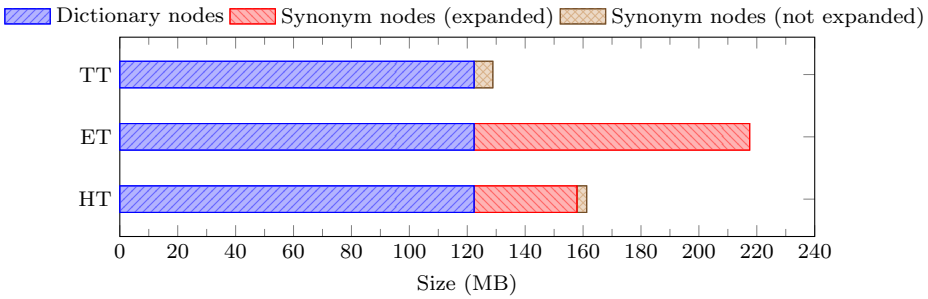
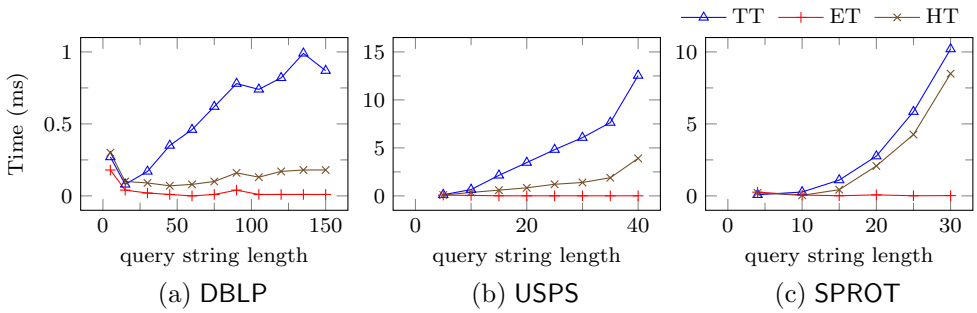


Figure 3.5: Space breakdown of proposed structures given SPROT dataset. DBLP and USPS have similar trends.

Space cost. The size of the constructed structure is depicted in Figure 3.5. TT is the obvious winner in this case since every dictionary string or synonym rule appears only one time in the structure. In contrast, ET uses nearly two times the space as TT, because, in ET, there are many synonym nodes attached to dictionary nodes. Finally, HT uses a moderate space: some frequent synonym rules are being expanded while others are left in a separate trie.

Section time. The final difference between the three methods is the speed of performing the top-k selection. We pick 50,000 random-length prefixes of random dictionary strings from each dataset as query strings, run our selection algorithms and summarise results in Figure 3.6. We can see that, although TT is fast to build and uses the least space, it has the longest selection time. This is because TT has to access the rule trie many times during the top-k process. On the other hand, ET performs the selection in a very short time since all synonym rules



(a) DBLP (b) USPS (c) SPROT

Figure 3.6: Time required for performing top-10 selections.

are already expanded, while HT has moderate performance because the most frequent synonym rules are already expanded hence less time for accessing the rule trie. Together with the aforementioned space cost, it is clear that HT is the best structure that strikes a balance between selection time and space cost.

Another finding is about the behaviour of HT: its speed is on a par with ET for DBLP and USPS but not SPROT. We found that the reason is the selected HT space threshold. Figure 8 in Paper I shows that, by increasing the space threshold to $0.75 \times (S_{ET} - S_{TT})$, HT can be 100% faster than the current setting.

3.4 Chapter summary

This chapter introduced Paper I, in which we proposed three methods to answer RQ1: use synonymical relevant words to enrich the quality of top-k selection tasks. Each proposed structure has a different optimisation goal: to minimise space cost (TT) or to accelerate top-k selections (ET). Furthermore, we propose HT that has a balanced space cost and selection speed, which makes it suitable for most application scenarios.

Chapter 4

Top-k similarity join with taxonomies

This chapter introduces Paper II that solves RQ2 and is organised as follows. First, it defines the research problem. Then, a basic list-based algorithm is introduced, followed by an optimised list-based and novel trie-based algorithms. Experimental results show that the latter two algorithms outperform the basic one to a significant extent.

4.1 Problem formalisation

Paper II defines RQ2 formally by using the taxonomy similarity measure $\mathrm{SIM_t}$. Given a positive integer k and two sets of strings $\mathcal{S}$, $\mathcal{T}$ in which each string maps to one taxonomy node in a given taxonomy hierarchy, find all pairs of strings in the form of $(S, T) \in \mathcal{S} \times \mathcal{T}$, such that $\mathrm{SIM_t}(S, T)$ is among the top k highest similarities of all pairs in $\mathcal{S} \times \mathcal{T}$.

As stated in Section 1.2, the above problem definition has a special case when one of $\mathcal{S}$ and $\mathcal{T}$ contains only one string, which is the query. In such a case, the above definition defines a top-k similarity selection problem.

4.2 Methods

Paper II contributes three algorithms, in which two are based on sorted lists, and one is based on the trie. We introduce them in Subsections 4.2.1 and 4.2.2.

Dewey labelling. To simplify the data representation, we use an existing scheme called *Dewey decimal system* [30] that assigns every tree node a label, henceforth every node can be represented by a Dewey label which is a series of node labels

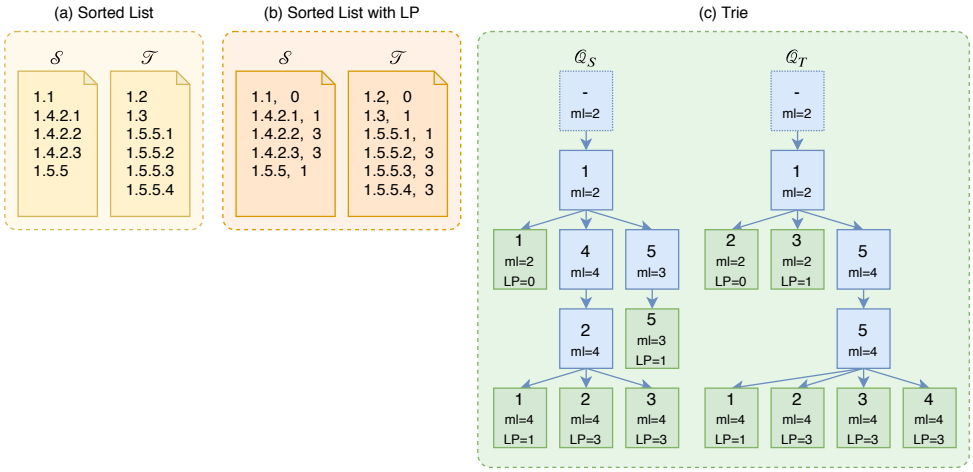


Figure 4.1: Illustration of proposed data structures. `minLen` is shortened to `ml` due to space limitation.

starting from the root to the node itself. Two Dewey labels can be compared in a lexicographical manner, e.g., 1.2 is less than 1.3 while 1.3 is greater than 1.2.3. The length of a Dewey label n, denoted by $|n|$, is the number of decimals separated by periods (if any), e.g., $|1| = 1$ and $|1.3.5| = 3$.

4.2.1 List-based algorithms

Basic algorithm (SortedTopK). Pre-sorting is one of the most common optimisations for join tasks. Given two sorted string lists (represented by matching taxonomy nodes, as in Figure 4.1a), the basic join algorithm is to use cursors to go through the list while maintaining the top-k highest-similarity results. The procedure can be summarised as follows:

1. Initialise a heap $\mathcal{P} : (S, T)$, where S and T are two strings, one from each input dataset. Items in the heap are sorted by $\mathrm{SIM_t}(S, T)$ in ascending order. Initialise a dynamic threshold $\theta = \mathrm{SIM_t}\left(\mathrm{peek}\left(\mathcal{P}\right)\right)$, i.e., equals the lowest similarity among pairs in the heap;

2. Initialise one cursor on $\mathcal{S}$ as C_1^1; and another cursor on $\mathcal{T}$ as C_2^1, pointing to the first Dewey string of each list;

3. Compare two Dewey strings at C_1^1 and C_2^1, assign the subscripts (1 or 2) to two variables min and max;

4. Initialise a forward cursor on the max list: $C^2_{\max} \leftarrow C^1_{\max}$;
5. While the first $\lceil \theta |C^1_{\min}| \rceil$ Dewey digits is the prefix of $C^2_{\max}$, do the following:
   (a) Verify the similarity of $(C^1_{\min}, C^2_{\max})$. If it is greater than or equal to θ, add $(C^1_{\min}, C^2_{\max})$ to $\mathcal{P}$;
   (b) If $\mathcal{P}$ contains more than k pairs, remove the head of $\mathcal{P}$ and update θ;
   (c) Advance $C^2_{\max}$.
6. Advance $C^1_{\min}$;
7. If none of C^1_1 and C^1_2 reaches the end of the corresponding list, go to Step 3;
8. Return all pairs in $\mathcal{P}$ and terminate.

Example 4.1 *We illustrate the basic algorithm with the example in Figure 4.1a. Let k be 2. Two pointers* C^1_1 *and* C^1_2 *are initially pointing to the first label of* $\mathcal{S}$ *and* $\mathcal{T}$*, respectively. By comparing* C^1_1 *and* C^1_2 *we get* $\min = 1$ *and* $\max = 2$*. At Step 5, since the first* $\lceil 0 \times 2 \rceil = 0$ *Dewey digits of* $C^1_{\min}$ *(1.1) is a prefix of* $C^2_{\max}$ *(1.2), the pair (1.1, 1.2) is being added to* $\mathcal{P}$ *and* θ *becomes 0.5. The algorithm then advances* $C^2_{\max}$ *(Step 5c) to read the next label 1.3, and thereafter (1.1, 1.3) is also added to* $\mathcal{P}$*. The algorithm moves on until* C^1_1 *and* C^1_2 *are at 1.5.5 and 1.5.5.1. Adding (1.5.5, 1.5.5.1) to* $\mathcal{P}$ *removes (1.1, 1.2) since the former has 0.75 similarity. Finally, only two of (1.5.5, 1.5.5.1), (1.5.5, 1.5.5.2), (1.5.5, 1.5.5.3), and (1.5.5, 1.5.5.4) remain in* $\mathcal{P}$*.*

The problem of the basic algorithm is the the number of LCA comparisons in Step 5a, where each comparison costs $O(\min\{|C^1_{\min}|, |C^2_{\max}|\})$ time. In the following optimised algorithm, we aim to reduce the number of such verifications.

Optimised algorithm (LP-SortedTopK). In the optimised algorithm, every Dewey string n is attached by an integer $\mathrm{LP}(n)$, which is the length of the LCA between n and the string before n. As an example, given a sorted list as in Figure 4.1b, $\mathrm{LP}(1.5.5.2) = 3$ since $\mathrm{LCA}(1.5.5.1, 1.5.5.2) = 3$.

The optimised algorithm makes uses of LP values to reduce the number of LCA comparisons. Specifically, it maintains two variables for the LCA length: global (x as in Algorithm 2 of Paper II) and local (y), where the global length equals $\mathrm{LCA}(C^1_1, C^1_2)$ and the local length equals $\mathrm{LCA}(C^1_{\min}, C^2_{\max})$. It is obvious that y can be used as LCA length when verifying the similarity. Besides, both lengths, as well as min and max, are updated by the following rules:

- If the local length y is greater than $\mathrm{LP}(C^2_{\max})$, decrease y to $\mathrm{LP}(C^2_{\max})$;
- If the global length x is greater than $\mathrm{LP}(C^1_{\min})$, decrease x to $\mathrm{LP}(C^1_{\min})$ and swap the values of min and max;
- If x is equal to $\mathrm{LP}(C^1_{\min})$, then recompute x, min, and max.

The above rules are proved in Lemma 3 of Paper II, and can be plugged into the basic algorithm to replace the Dewey comparison in Step 3 and the similarity calculation in Step 5a. After that, only in one of three cases the LCA length needs to be recomputed.

4.2.2 Trie-based algorithm

The previous list-based algorithms will always access every string, regardless of whether the access is only to get its LP value. To further improve the join performance, Paper II designs a trie-based algorithm, named TrieTopK, that avoids accessing unfeasible items entirely.

Data structure. We build two tries for $\mathcal{S}$ and $\mathcal{T}$ as $\mathcal{Q}_S$ and $\mathcal{Q}_T$. In addition to the properties of a standard trie stated in Subsection 2.3.1, we add two more properties to each node:

- Each node holds an integer, $\mathrm{minLen}(n)$, which indicates the smallest depth among all its flagged descendants. The minLen of a flagged node is equal to the depth of the node itself.
- Each flagged node holds an LP value, which is the same as in the previous list-based algorithms.

Figure 4.1c illustrates an example of the trie structure.

Maximal sub-trie similarity. Given one trie node n that appears in both $\mathcal{Q}_S$ and $\mathcal{Q}_T$. Let $\mathcal{Q}'_S$ and $\mathcal{Q}'_T$ be two sub-tries with root n, and let N_S and N_T be two collections of flagged nodes in $\mathcal{Q}'_S$ and $\mathcal{Q}'_T$. Paper II formulates a maximal sub-trie similarity (MSS) as the highest similarity one can obtain by pairing up nodes from N_S and N_T. The MSS has two properties:

- $\mathrm{MSS}(n, N_S, N_T)$ is at its maximum when $N_S = \mathcal{Q}'_S$ and $N_T = \mathcal{Q}'_T$, and is monotonically decreasing when removing nodes from N_S and/or N_T;

- The maximum of $\text{MSS}(n, N_S, N_T)$ can be calculated by using the value of minLen of n in two tries:

$$\text{MSS}_{\text{init}}(n) = \frac{|n|}{\max\left(\text{minLen}\left(n, \mathcal{Q}'_S\right), \text{minLen}\left(n, \mathcal{Q}'_T\right)\right)},$$

  where $\text{minLen}(n, \mathcal{Q})$ returns the minLen of a node from $\mathcal{Q}$ that holds a Dewey label same as that of n.

The algorithm. The trie-based algorithm, as in Algorithm 4 of Paper II, utilises these properties to eliminate unnecessary node accesses. Specifically, for every common node n between $\mathcal{Q}_S$ and $\mathcal{Q}_T$, we calculate its MSS_{init} and add n to a max heap sorted by MSS. Then, the algorithm continuously polls the head node n_c from the heap and traverses two sub-tries $\mathcal{Q}'_S$ and $\mathcal{Q}'_T$ in breadth-first (BFS) manner, and pairing up all flagged nodes it encountered. Such a process continues until the next encountered pair has a similarity less than the MSS of the new heap head (since the previous head has been polled out). When this happens, the algorithm decreases the MSS of n_c to the similarity of the next pair, and adds n_c back to the heap. Note that n_c will not be the head item anymore since its MSS is now decreased, and thus will not be accessed again until no other node in the heap has a higher MSS.

The correctness of the trie-based algorithm depends on two observations: (i) if the current heap head is n, then no pair from $N_S \times N_T$ can have a higher similarity than $\text{MSS}(n, N_S, N_T)$; and (ii) when traversing $\mathcal{Q}'_S$ and $\mathcal{Q}'_T$ using BFS, the similarity of any visited node pair is higher than that of any unvisited pair. Paper II proves both observations in Theorem 6.

4.3 Experimental results

This section presents some important experimental results of Paper II. We use join time and scalability metrics to compare the basic against proposed list- and trie-based algorithms.

Dataset and setup. For experiments, we employ two real taxonomy trees: Medical Subject Headings[1] (MeSH) and Wikipedia categories[2] (WIKI), each contains hierarchical IS-A relations such as "Nature → Energy → Energy conversion → Hydro-power" (with Dewey label "12945.7.10.18"). The sizes of these datasets are

[1] `https://www.nlm.nih.gov/mesh`

[2] `https://wiki.dbpedia.org/develop/datasets`

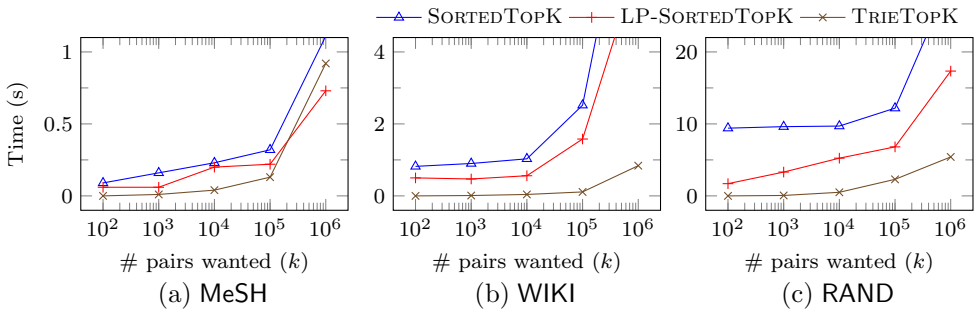


Figure 4.2: Running time of top-k algorithms w.r.t. k.

57,840 and 1,212,943, respectively. Apart from real data, we also generated a large synthetic dataset (RAND) which consists of 10,000,000 records. All algorithms are implemented in Java 8 run on a high-performance computing node which has two quad-core Intel Xeon 2.53GHz CPU, 32GB RAM, and Ubuntu as the operating system.

Join time. The first test is about the response time for returning the top-k similar pairs. To simulate different usage scenarios, we employs fives k's: 10^2, 10^3, 10^4, 10^5, and 10^6. The result plotted in Figure 4.2 shows that all three algorithms have their response times increased as k becomes larger, but with different starting points and rates. As an example, the response times for returning the top-100,000 pairs given the large WIKI dataset are 2.52, 1.58, and 0.11 seconds for SortedTopK, LP-SortedTopK, and TrieTopK, respectively, indicating a large performance gap. TrieTopK is the best algorithm for all use scenarios by having 5 seconds of running time, even on the huge RAND dataset.

Scalability. Finally, we draw Figure 4.3 to visualise the scalability of top-k algorithms by asking for the top-20,000 most similar record pairs. We have observed that data-size is a key factor influencing the top-k performance. For example, all three algorithms finish within 0.2 seconds on a small dataset (MeSH), and require 10 seconds on huge datasets such as RAND. Among the three algorithms, SortedTopK runs the slowest, followed by LP-SortedTopK and TrieTopK, which is due to the number of LCA comparisons. Specifically, TrieTopK can maintain very high performance on all three datasets, as it does not access unfeasible nodes at all.

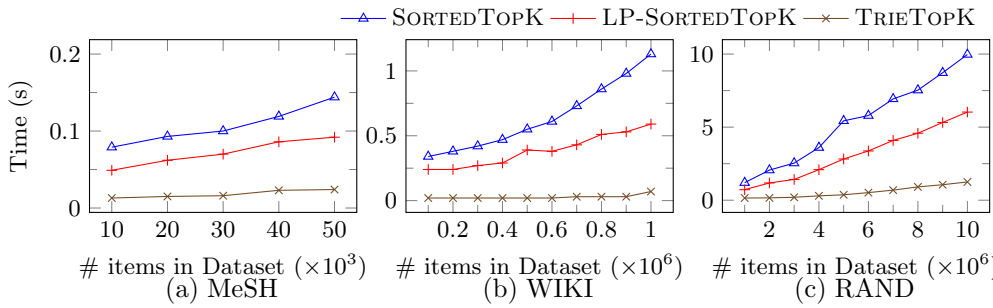


Figure 4.3: Scalability of top-k algorithms when $k = 20,000$.

4.4 Chapter summary

This chapter introduced Paper II, in which we proposed three algorithms to solve RQ2: select the top-k similar string pairs w.r.t. taxonomy knowledge. Main contributions in this paper include (i) a list-based algorithm which pre-computes LCA to avoid slow similarity computation and (ii) a trie-based algorithm that eliminates unnecessary node accesses to speed up the join process.

Chapter 5

All-match similarity join with taxonomies

In this chapter, we give an overview of the results of Paper III aiming to solve RQ3. First, we present the similarity measure and problem formalisation, then describe a novel prefix filtering algorithm that finds similar string pairs efficiently. Next, we introduce a suggestion mechanism which automatically finds the optimal parameter value, followed by presenting experimental results depicting the improvements over the state of the art.

5.1 Problem formalisation

Similarity measure. Paper III assumes that each input string in the join problem can be split into one or more "segments", each of which maps to a node in a given taxonomy hierarchy. Formally, two strings now form two node collections: $S = \{s_1, s_2, \cdots, s_i\} \subseteq \mathcal{Q}$ and $T = \{t_1, t_2, \cdots, t_j\} \subseteq \mathcal{Q}$, where $\mathcal{Q}$ is a taxonomy. The similarity between two arbitrary nodes (s, t) an be obtained from Equation 2.3 as $\mathrm{sim_t}(s, t)$. Subsequently, we calculate the overall similarity between S and T from a bipartite matching model as the maximal similarity sum:

$$\mathrm{SIM_t}(S,T) = \frac{W(S,T)}{\max(|S|,|T|)} = \frac{\max \sum_p \sum_q x_{pq} \mathrm{sim_t}(s_p, t_q)}{\max(|S|,|T|)} \tag{5.1}$$

where $p \in [1, i]$, $q \in [1, j]$, an indicator $x_{pq} \in \{0, 1\}$, $\sum_p x_{pq} \leq 1$, and $\sum_q x_{pq} \leq 1$. W in the above equation is the bipartite matching model, which is solvable in $O(n^3)$ time by using the Hungarian algorithm [31].

Problem formalisation. The research problem of Paper III is defined based on the similarity measure $\mathrm{SIM_t}$. Given a value $\theta \in (0, 1]$ and two sets of strings $\mathcal{S}$, $\mathcal{T}$, in which each string maps to one or more taxonomy nodes in a given taxonomy hierarchy, find all pairs of strings in the form of (S, T), $S \in \mathcal{S}$, $T \in \mathcal{T}$, such that $\mathrm{SIM_t}(S, T) \geq \theta$.

5.2 Adaptive prefix filtering

To avoid calculating $\mathrm{SIM_t}$ for all possible strings pairs (i.e., a nested loop join), Paper III proposed a new prefix filtering framework to exclude some unfeasible pairs from the similarity calculation. The highlight here is that, instead of searching for pairs with at least one overlapped prefix, the new method allows finding more than one of them to reduce further the number of false positives (i.e., to have less potentially-similar pairs). Lemma 5.1 guarantees the correctness of this method.

Lemma 5.1 *Given S and T as two strings mapped to taxonomy nodes, and without loss of generality by assuming $|S| < |T|$. If $SIM_t(S, T) \geq \theta$, then there are at least τ distinct node pairs in the form of (s, t) such that each of them satisfies $sim_t(s, t) \geq \phi$, where $\phi = \frac{\theta|T|-\tau+1}{|S|-\tau+1}$.*

Lemma 5.1 states a necessary but not sufficient condition for S and T being similar. In the context of taxonomy similarity, the task of finding "τ distinct node pairs" can be achieved by finding pairs of nodes with *common ancestors*. One efficient way to do this is by using *inverted lists*.

Two optimisations are available when constructing inverted lists. First, for each input string list, one can relax Lemma 5.1 by replacing "$|T|$" by "$|S|$". Next, for an arbitrary node s in a string S, it is only needed to add ancestors with depth at least $\phi|s|$ into the inverted list, as shallower ancestors are insufficient to contribute ϕ similarity to the string. Such optimisations reduce the index size thus accelerating the filtering.

Join algorithm (AP-Join). The join procedure becomes straightforward after two inverted lists are constructed. The pseudocode is in Algorithm 1 of Paper III, while we list each step in the following:

1. Join two inverted lists to find all strings indexed by the same key;
2. Go through all string pairs, and for each string pair, count the number of their common keys. Mark the pair as a candidate if the number reaches τ;

3. Calculate (verify) the similarity $\mathrm{SIM_t}$ for each candidate pair. If the similarity reaches θ, then add such a candidate to the output.

5.3 Parameter selection

In Lemma 5.1, the most important parameter that affects the performance is τ. Intuitively, a small τ leads to smaller inverted lists thus faster filtering, but increases the verification time due to more candidates caused by the loosened overlap constraint. On the other hand, a large τ inflates the inverted list but deflates the candidate set. Hence, finding the balance between filtering and verification times becomes critical towards the minimised join time.

In Paper III, we designed an estimation procedure to predict the join time of a given τ using small samples. The procedure begins by building a cost model as

$$C_\tau = C_{F_\tau} + C_{V_\tau} = t_F F_\tau + t_V V_\tau \tag{5.2}$$

where C_τ is the total join cost (i.e., running time) consisting of filtering (C_{F_τ}) and verification (C_{V_τ}), each of which is obtained by multiplying the time for processing one pair (t_F for filtering and t_V for verification) by the number of pairs to be processed (F_τ and V_τ).

To estimate F_τ and V_τ for huge datasets, we employ the *independent Bernoulli sampling* [45] method which picks each string in $\mathcal{S}$ and $\mathcal{T}$ by probabilities p_s and p_t. Then, a candidate or result pair (S, T) will be in the sample iff both S and T are picked. We hence obtained two unbiased estimators:

$$\hat{F}_\tau = \frac{F'_\tau}{p_s p_t}, \qquad \hat{V}_\tau = \frac{V'_\tau}{p_s p_t}$$

where F'_τ and V'_τ is obtained by running the join algorithm on two samples. An estimator for the total join cost, $\hat{C}_\tau$, is obtained subsequently by plugging $\hat{F}_\tau$ and $\hat{V}_\tau$ into Equation 5.2.

Iterative suggestion refinement. In the above method, the sample size should be sufficiently large to ensure accurate estimations. However, considering the nature of join operations, the maximal number of operations is bounded by the sample size $|S'| \times |T'|$, which will increase significantly as the sample size increases. To reduce the number of calculations, Paper III introduces an "iterative" method which improves the estimation accuracy not by increasing the sample size, but by using small samples and more estimation iterations. Such a refinement process

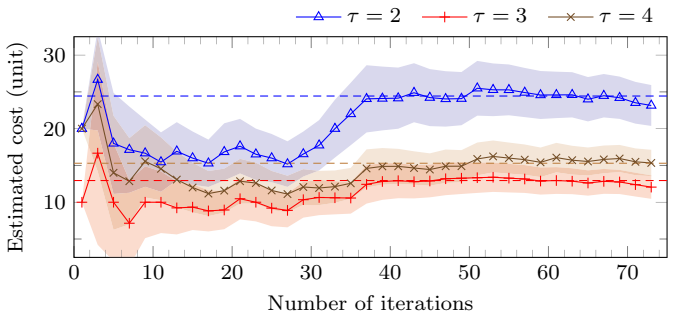


Figure 5.1: Illustration of the refinement process. Solid lines are estimated means, shaded areas are confidence intervals, dashed lines are empirical real costs.

goes on until the estimation is "accurate enough", determined by its confidence interval (CI) [28].

The refinement process is backed by the observation that each estimation, $\hat{F}_\tau$ or $\hat{V}_\tau$, on a small sample is an independent estimation of the real value F_τ or V_τ. Therefore, according to the central limit theorem (CLT), by having a series of estimations $\{\hat{F}_\tau^{(1)}, \hat{F}_\tau^{(2)}, \cdots\}$ and $\{\hat{V}_\tau^{(1)}, \hat{V}_\tau^{(2)}, \cdots\}$, it is possible to use their means $\mu_{\hat{F}_\tau}$ and $\mu_{\hat{V}_\tau}$ as the estimations of F_τ and V_τ. The total cost C_τ can be estimated subsequently. Figure 5.1 illustrates the refinement process on a real dataset.

The refinement process stops when the τ leading to the minimal total cost is identified with high confidence. Specifically, given a universe $\mathbb{T}$, the stop criterion is when the overlap between CIs of (i) the smallest C_{τ_1} and (ii) all other C_{τ_2}'s, $\tau_2 \in \mathbb{T} \setminus \{\tau_1\}$, are small enough, as stated in Lemma 4.1 of Paper III.

5.4 Experimental results

This section highlights the performance of our proposed AP-Join algorithm. As for the baseline, we choose the state-of-the-art K-Join algorithm [38], which is a prefix filtering principle with one overlapped prefix. We obtained the source code from its authors.

Dataset and setup. The experimental section of Paper III employs two taxonomy datasets as knowledge: Wikipedia categories[1] and MeSH medical terms[2], consisting of 1,212,943 and 57,840 nodes, respectively. For the join task, it uses

[1] `https://en.wikipedia.org/wiki/Help:Category`
[2] `https://www.nlm.nih.gov/mesh/`

WIKI page titles[3] (3,512,954 records) and OHSUMED paper headings[4] (293,294 records). We ran our algorithms on a Ubuntu computer with a Xeon 2.53GHz CPU and 32GB RAM.

Join time. The first important result is about the join time. As shown in Table 5.1, our algorithm AP-Join can significantly reduce the number of candidates and therefore shortened the total join time. It is also worth noticing that K-Join generates smaller inverted lists for the OHSUMED dataset, which confirms the analysis regarding different τ's in Section 5.3, since the K-Join's strategy is equivalent to our algorithm with $\tau = 1$.

Suggestion time and accuracy. Another important result is regarding the parameter suggestion process. To form Table 5.2, we run the suggestion algorithm 128 times for each θ and count the number of correct suggestions (according to the empirical knowledge). The result shows that the suggestion algorithm has a high 90% accuracy on two datasets. The right half of Table 5.2 also shows that the estimation procedure occupies only a tiny amount of time from the join process, which is less than three seconds on the WIKI and five seconds on the OHSUMED datasets.

5.5 Chapter summary

This chapter introduced Paper III, in which we proposed a filtering and verification framework to solve RQ3: taxonomy-based string similarity joins. Contributions include a new prefix filtering principle that allows the uses of multiple overlaps to reduce candidate size, as well as an estimation algorithm to suggest the best number of overlaps which leads to the shortest join time.

[3] `https://wiki.dbpedia.org/`

[4] `https://trec.nist.gov/data/t9_filtering.html`

Table 5.1: Join performance of the proposed algorithm (AP-Join) vs the state of the art (K-Join). "OOT" stands for "out of time".

Dataset	Algorithm	# of Pairs (10^8)			# of Candidates (10^6)			Running time (min)		
		θ: 0.6	0.7	0.8	0.6	0.7	0.8	0.6	0.7	0.8
WIKI	AP-Join	**0.42**	**0.08**	**0.01**	**11.52**	**3.04**	**0.27**	**10.03**	**2.64**	**0.55**
(50K×50K)	K-Join	0.87	0.25	0.07	28.42	8.35	1.98	22.28	6.65	1.74
OHSUMED	AP-Join	**1.08**	4.97	1.72	**63.43**	**0.64**	**0.26**	**41.81**	**4.44**	**1.67**
(50K×50K)	K-Join	OOT	**2.13**	**0.86**	OOT	115.58	38.42	OOT	80.01	25.33

Table 5.2: Performance of the parameter suggestion procedure. Sample size is 100, confidence level is set to 70% on both sides.

Dataset	Accuracy from 128 runs				Estimation time (s)			
	θ: 0.6	0.7	0.8	0.9	0.6	0.7	0.8	0.9
WIKI	92.03%	100%	100%	100%	2.58	1.69	1.74	1.51
OHSUMED	96.09%	99.22%	90.63%	100%	4.63	1.10	2.04	1.42

Chapter 6

All-match similarity join with a unified similarity

This chapter overviews the contributions of Paper IV for solving RQ4: measuring string similarity concerning multiple types of similarities and performing join operations. It starts by introducing a unified measure that is capable to take multiple types of similarities, followed by a transformation showing the hardness of finding the optimal solution. Two approximate filtering algorithms are proposed, aiming for a low false-positive rate and a high filtering speed, respectively.

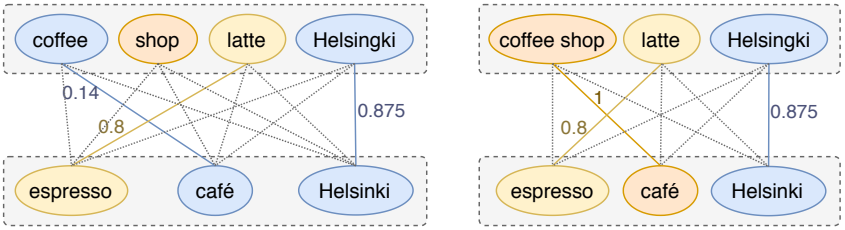
6.1 Problem formalisation

Paper IV employs multiple similarity measures, namely Jaccard (for typographic errors), synonym, and taxonomy similarities, to find the highest similarity for two strings. As a high level overview, it breaks strings into small *segments* (i.e., substrings containing one or more words, recall Section 5.1), choose the best similarity measure for each two segments, and then aggregate all segments' similarity to obtain the overall string similarity.

6.1.1 The unified similarity measure

We now define the new similarity measure. First, given two strings S and T divided into segments, let P_S (P_T) denote an arbitrary segment from S (T). By applying Jaccard, synonym, and taxonomy similarities onto (P_S, P_T), one can obtain the maximum, denoted by msim:

$$\text{msim}(P_S, P_T) = \max\{\text{sim}_\text{j}(P_S, P_T), \text{sim}_\text{s}(P_S, P_T), \text{sim}_\text{t}(P_S, P_T)\}. \tag{6.1}$$



(a) Separated "coffee" and "shop". Similarity is (0.14+0.8+0.875)/4 = 0.454 by assuming 'e'='é'

(b) Combined "coffee" and "shop". Similarity is (1+0.8+0.875)/3 = 0.892

Figure 6.1: Illustration of how partitions affect the similarity, using the same knowledge as in Figure 1.1.

Next, let $\mathcal{P}_S = \{P_1, P_2, \cdots, P_i\}$ ($\mathcal{P}_T$) be a collection of segments of S (T), such that each segment in $\mathcal{P}_S$ ($\mathcal{P}_T$) maps to a taxonomy node, synonym rule, or a single word (for Jaccard measure). Given a pair of partitions $(\mathcal{P}_S, \mathcal{P}_T)$, their similarity can be defined as follows, similar to SIM_t from Section 5.1:

$$\text{SIM}_\text{u}(\mathcal{P}_S, \mathcal{P}_T) = \frac{\max \sum_{i=1}^{|\mathcal{P}_S|} \sum_{j=1}^{|\mathcal{P}_T|} x_{ij} \cdot \text{msim}\big(P_{S_i}, P_{T_j}\big)}{\max\{|\mathcal{P}_S|, |\mathcal{P}_T|\}},$$
$$\text{where } x_{ij} \in \{0,1\}, \sum_i x_{ij} \leq 1, \text{and } \sum_j x_{ij} \leq 1. \tag{6.2}$$

Since there is more than one way to partition strings (see Figure 6.1 for an example), the *unified similarity* between two strings (S, T) becomes the maximal similarity among all pairs of possible partitions $(\mathcal{P}_S, \mathcal{P}_T)$:

$$\text{USIM}(S, T) = \max_{\forall(\mathcal{P}_S, \mathcal{P}_T)} \{\text{SIM}_\text{u}(\mathcal{P}_S, \mathcal{P}_T)\}. \tag{6.3}$$

Paper IV proved the NP-hardness of calculating USIM by presenting a reduction from USIM to a well-known NP-hard problem: *weighted maximum independent set* (w-MIS) [2, 5, 22]. To this end, Paper IV designed an approximation algorithm with a non-trivial bound and runs in a polynomial time.

6.1.2 Approximation algorithm

The proposed approximation algorithm works by transforming the calculation of USIM into a problem that selects vertices of a weighted graph towards the highest weight sum. The transformation process is: (i) corresponding to each

possible segment pairs (P_S, P_T), add one vertex to the graph and set its weight to $\text{msim}(P_S, P_T)$; and (ii) for each *pair of segment pairs* that contain the same word, draw an edge between corresponding vertices in the graph. The goal is to find a set of independent vertices with the maximised weight sum, which is proven (in Paper IV) equivalent to finding the maximum of the numerator of SIM_u.

Let a *d-claw* be a induced sub-graph that consists of d independent vertices, called *talons*, and one centre vertex that connects to all d talons. Let k be the maximal number of words in any segment pair, and then, we claim that the graph obtained from the aforementioned transformation procedure is "k+1-claw free", meaning that any claw in the graph can have at most k talons. This observation enables us to approximate the w-MIS of such a graph in a polynomial time [7, 20]. Many approximation algorithms have been proposed in the past decades [3, 5, 7, 10, 20], among which we adapt the state of the art, SquareImp [7], to find the w-MIS of the graph and, in turn, solve our USIM problem with an approximation ratio $\frac{t}{t-1} \cdot \frac{k^2-1}{2}$, where $t > 1$. Our procedure is to combine the SquareImp algorithm (Steps 1.1 and 1.2) with an additional heuristic (Steps 2.1 and 2.2):

1.1. Continuously finding a claw that can improve at least $\frac{1}{t}$ the sum of squared weights of selected vertices;

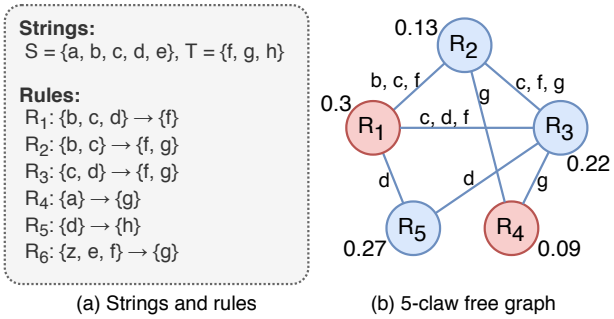
1.2. For each such claw, select all its talons and un-select all vertices connected to the talons;

2.1. Continuously finding a claw that can improve at least $\frac{1}{t}$ the similarity SIM_u of selected vertices;

2.2. For each such claw, select all its talons and un-select all vertices connected to the talons.

The approximation ratio consists of two parts: (i) the original bound of SquareImp, which is $\frac{t}{t-1} \cdot \frac{k+1}{2}$, and (ii) the bound of the denominator of SIM_u, which is proved $k - 1$ in the worst case.

Example 6.1 *Given two strings and synonym rules in Figure 6.2a. We construct a 5-claw-free graph in Figure 6.2b. SquareImp selects* R_2 *and* R_5*, and hence two partitions* $\mathcal{P}_S = \big\{\{a\}, \{b, c\}, \{d\}, \{e\}\big\}$, $\mathcal{P}_T = \big\{\{f, g\}, \{h\}\big\}$ *leading to a similarity* $\frac{0.13+0.27}{4} = 0.1$*. Then, the heuristic in Steps 2.1 and 2.2 finds another claw with centre* R_2 *and talons* R_1, R_4 *can improve the similarity by partitioning two strings as* $\mathcal{P}_S = \big\{\{a\}, \{b, c, d\}, \{e\}\big\}$, $\mathcal{P}_T = \big\{\{f\}, \{h\}, \{g\}\big\}$ *which results in a similarity* $\frac{0.3+0.09}{3} = 0.13$*. The final result is* 0.13*.*



(a) Strings and rules (b) 5-claw free graph

Figure 6.2: Illustration of Example 6.1. Numbers beside each vertex is its weight, letters beside each edge are words that appear in both rules. SquareImp first selects R_2 and R_5; the final answer is R_1 and R_4.

6.1.3 Problem formalisation

The research problem RQ4 can be formalised as follows. Given a value $\theta \in (0, 1]$ and two lists of strings $\mathcal{S}$, $\mathcal{T}$ in which words in each string map to either taxonomy nodes, synonym rules, or contain typos. The task is to find all pairs of strings in the form of $(S, T) \in \mathcal{S} \times \mathcal{T}$, such that $\text{USIM}(S, T) \geq \theta$.

6.2 Prefix filtering

Paper IV introduces a prefix filtering mechanism to select string pairs that are *potentially* similar. As a high-level view, by representing various types of similarities using a unified structure (called *pebbles*), the prefix filtering can select some of them to become the "prefix" of the corresponding string. Strings with enough common prefixes are potentially similar and are verified.

Pebble. Pebble, denoted by B, is a structure that holds a portion of a segment and a number as weight, denoted by $\text{w}(B)$. Given a segment P mapping to a similarity measure, its pebbles are generated in different ways:

- Jaccard coefficient: pebbles are all q-grams of P. The weight of each B equals $\frac{1}{|\text{grams}_q(P)|}$;
- Synonym similarity: assume that the applicable synonym rule of P is R, then the pebble B of P is $\text{lhs}(R)$, with a weight $\text{w}(B) = \text{C}(R)$;

- Taxonomy similarity: let n be the matching taxonomy entity of P. Pebbles of P are n and all its ancestors, each has a weight $\mathrm{w}(B) = \frac{1}{\mathrm{depth}(n)}$.

Example 6.2 *Given two strings "coffee" and "cafe", as well as three types of knowledge as in Figure 1.1. Pebbles of "coffee" are (i) five* 2*-grams w.r.t. Jaccard coefficient:* {*co, of, ff, fe, ee*}*, each weights* $\frac{1}{5}$*; and (ii) three taxonomy entities w.r.t. taxonomy similarity:* {*Wikipedia, food, coffee*}*, each weights* $\frac{1}{3}$*. Pebbles of "cafe" are (i) three* 2*-grams:* {*ca, af, fe*}*, each weights* $\frac{1}{3}$*; and (ii) one synonym entity:* {*coffee shop*}*, weights* 1*.*

Prefix filtering. After obtaining pebbles of all segments of a given string, one can apply a *weighted* prefix filtering principle to select some pebbles as the signature used for the actual filtering stage. Briefly speaking, the procedure consists of the following steps:

1. For each input string S, calculate the lower bound, $\mathrm{LB}(S)$, of the weight sum of pebbles in its signature. This is done by multiplying the similarity threshold θ and the minimum of the number of segments among any possible partition of S, which is estimated by solving a *minimum exact cover* problem [22];
2. Sort all pebbles by global frequency in ascending order (i.e., IDF [37, 38]);
3. The algorithm removes the last pebble from the sorted list. It meanwhile tracks, for each segment and each similarity measure, the sum of weights of all removed pebbles. Then:
   (a) For each segment, identify the maximum among all weight sums. If the maximums of all segments add up to $\mathrm{LB}(S)$, it puts the current pebble back on the list and goes to Step 4;
   (b) Otherwise, go to Step 3 to process the next pebble.
4. Return the remaining pebbles in the sorted list as the signature of S.

After obtaining the signatures of all strings, a filtering procedure starts to find all string pairs with at least one common pebble (as described in Subsection 2.3.2) and after that send them out for verification, as described in Subsection 6.1.1.

Example 6.3 *Given three types of knowledge and measure as in Figure 1.1,* $\theta =$ 0.8 *and the string* T *of 3 segments* P_1*="espresso",* P_2*="cafe", and* P_3*="Helsinki".*

Table 6.1: Illustration to Example 6.3. Columns J, T, and S are the sum of weights of removed pebbles w.r.t. to different similarity measures: Jaccard, Taxonomy, and Synonym. Underlined numbers are those changed in each step, numbers in bold are being accumulated into the removed weight sum.

i	Pebble	P_1: "espresso"			P_2: "cafe"			P_3: "Helsinki"			Removed weight sum	Remaining $\tau - 1$ sum
		J	T	S	J	T	S	J	T	S		
23	es	**1/7**									0.143	1.667
22	ss	**2/7**									0.286	1.667
21	fe	**2/7**			**1/3**						0.619	1.667
20	Wikipedia	**2/7**	1/5		**1/3**						0.675	1.667
19	food	2/7	**2/5**		**1/3**						0.733	1.667
...	...	...	...		...			...			...	...
8	ca	4/7	**1**		**2/3**			**5/8**			2.292	1.476
7	ki	4/7	**1**		**2/3**			**6/8**			2.417	1.476

The lower bound $LB(T) = \theta \times \lceil \frac{3}{\ln 1+1} \rceil = 0.8 \times 3 = 2.4$*, where* 3 *is the number of selected segments by the greedy algorithm and* $\ln 1 + 1$ *is the approximation bound. T generates 23 pebbles. Next, as illustrated in Table 6.1, the algorithm removes the 23rd pebble from the list and accumulate* $\frac{1}{7}$ *to the value of removed Jaccard weight. The procedure continues to calculate the removed weight by summing up the highest weight corresponding to each segment (numbers in bold in the table) until it reaches* 2.4*. The algorithm finally stops at the 7th pebble, puts it back on the list, and returns the remaining 7 pebbles as the signature of T.*

6.2.1 Adaptive prefix filtering

The performance of the above prefix filtering algorithm is sensitive to the value of overlap threshold, which is the same as for the taxonomical join in Section 5.2. To allow users to tweak the algorithm, Paper IV proposes two signature selection scheme to support finding > 1 common prefixes: one fast heuristic and one precise dynamic programming scheme.

The heuristic is based on an idea that one can only remove pebbles such that there must be at least $\tau \geq 1$ more pebbles to make the total weight reach the lower bound $\text{LB}(S)$. The procedure is a modification of the previous prefix filtering algorithm, by replacing Step 3a by the following:

> *If the sum of (i) the current "removed weight sum of all segments" and (ii) the sum of weights of the top* $\tau - 1$ *heaviest remaining pebbles reaches LB(S), put the current pebble back on the list and go to Step 4.*

Example 6.4 *Recall Example 6.3 but this time with the heuristic approach assuming* $\tau = 4$*. When* $i = 19$*, the top* $\tau - 1 = 3$ *heaviest remaining pebbles are "coffee shop" (the lhs of synonym rule of* P_2*, weights* 1*) and two grams "ca", "af" (each weights* $\frac{2}{3}$*). The 19th pebble cannot be removed because* $0.733 + 1 + \frac{2}{3} = 2.4$*, no less than LB(T)* $= 2.4$*. The algorithm returns the remaining 19 pebbles as the signature of* T*.*

Dynamic programming (DP). The heuristic, as mentioned earlier, runs in a linearithmic time but does not always return the shortest signature because the weight sum of the remaining $\tau - 1$ heaviest pebbles is not tight. To get an accurate sum and thereby shorter signature, Paper IV proposes a second prefix selection algorithm which utilises dynamic programming.

Given an integer τ, a string S which contains t segments, and an integer i indicates that we are testing whether the i-th pebble can be removed. The subproblem of the dynamic programming algorithm is defined as follows:

> *For each* $p \in [0, t]$ *and* $d \in [0, \tau - 1]$*, solve* $\mathbb{W}_i[p, d]$ *which is the maximal increment of weight sum of removed pebbles, by removing* d *more remaining pebbles (i.e., from the 1st to the* $(n-1)$*-th pebbles), where all such pebbles are from the first* p *segments of* S*.*

When it arrived $p = t$ and $d = \tau - 1$, the value of $\mathbb{W}_i[t, \tau - 1]$ becomes the maximal weight increment by removing $\tau - 1$ remaining pebbles of all segments. If the sum of $\mathbb{W}_i[t, \tau - 1]$ and the "removed weight sum of all segments" (accumulated the by same method from the previous section) reaches $\mathrm{LB}(S)$, then the i-th pebbles can not be removed from S's signature.

The detailed algorithm can be found in Algorithm 5 in Paper IV, in which we use an accessory table $\mathbb{V}_i[p, c]$, $c \leq d$, to store the maximal weight increment by adding c new pebbles from only the p-th segment. When calculating $\mathbb{W}_i[p, d]$, the algorithm finds the maximum among all d options, i.e., $\mathbb{W}_i[p, d] = \max_{c \in [0,d]} \mathbb{W}_i[p - 1, d - c] + \mathbb{V}_i[p, c]$.

Example 6.5 *Recall the setting in Example 6.4, where* $\theta = 0.8$*,* $\tau = 4$*, and* $i = 19$*. Table 6.2 illustrates the DP table* $\mathbb{W}_{19}$ *and the accessory table* $\mathbb{V}_{19}$*. For example,* $\mathbb{V}_{19}[1, 3]$ *is the maximal similarity increment of removing three remaining taxonomy pebbles of* P_1*, i.e.,* $\frac{5}{5} - \frac{2}{5}$*;* $\mathbb{W}_{19}[2, 3]$ *is obtained as* $\mathbb{W}_{19}[1, 2] + \mathbb{V}_{19}[2, 1]$*, i.e., removing two pebbles from* P_1 *and one from* P_2*. In the end, the sum of* $\mathbb{W}_{19}[3, 3]$ *and the removed weight sum is* $1.067 + 0.733 = 1.800 < LB(T)$*. This concludes that the 19th pebble can be safely removed from* T*'s signature.*

Table 6.2: Illustrating Example 6.5, the DP prefix selection method. $i = 19$. Settings are the same as in Example 6.4.

p	Segment	DP table $\mathbb{W}_{19}$				Accessory table $\mathbb{V}_{19}$			
		d: 0	1	2	3	c: 0	1	2	3
0	-	0	0	0	0	-	-	-	-
1	P_1: "espresso"	0	0.2	0.4	0.6	0	3/5-2/5	4/5-2/5	5/5-2/5
2	P_2: "cafe"	0	0.667	0.867	1.067	0	1-1/3	1-1/3	1-1/3
3	P_3: "Helsinki"	0	0.667	0.867	1.067	0	1/8-0	2/8-0	3/8-0

6.3 Join algorithm and parameter suggestion

The join procedure is presented in Algorithm 6 of Paper IV. The procedure consists of four steps:

1. Generate signatures of all strings using either the heuristic or the DP. Build one inverted list for each input list;
2. Join two inverted lists to find all strings indexed by the same key;
3. Go through all string pairs, count the total number of common keys of each pair. If the number of common keys is at least τ, mark the pair as a candidate;
4. Go through all candidates, calculate the similarity USIM$(\cdot)$ of each candidate pair by using the approximation algorithm presented in Subsection 6.1.1. If it reaches the given threshold θ, add this candidate to the output.

The critical decision in the join algorithm is to decide the value of τ. Nevertheless, Paper IV proves that the value of τ affects the length of signatures and number of candidates, and subsequently influence the filtering and verification speed. To this end, Paper IV designed a recommendation algorithm similar to that in Section 5.3. Specifically, the algorithm is based on a cost model and multiple stages of independent Bernoulli sampling. The final recommendation becomes ready when overlaps of CIs of different τ's are small enough.

6.4 Experimental results

This section presents some important experimental results of Paper IV. Specifically, we show the join time, filtering power, scalability of all proposed algorithms, and

Table 6.3: Characteristics of used taxonomies and synonyms.

Taxonomy (Height in min/avg/max)				Synonym	
Source	# of nodes	Height	Average fanout	Source	# of rules
MeSH tree	57,840	1 / 5.1 / 12	157	Aliases	180,259
Wiki categories	1,212,943	1 / 6.2 / 26	32,300	Synonyms	680,625

Table 6.4: Characteristics of used string datasets.

Source	# of strings	... per string (min/avg/max)			
		Characters	Words	Taxonomies	Synonyms
MED	293,294	2 / 110.5 / 452	1 / 8.4 / 26	0 / 3.2 / 18	0 / 4.3 / 15
WIKI	3,512,954	2 / 161.5 / 8,588	1 / 8.2 / 277	0 / 6.2 / 185	0 / 2.0 / 98

compare our algorithm to three state-of-the-art algorithms combined. The results show the superiority of AU-Filter in finding similar strings considering multiple types of similarities.

Dataset and setup. Tables 6.3 and 6.4 summarise the datasets used for the experiments. For knowledge, there are (i) two taxonomies: MeSH tree[1] and Wikipedia categories, containing hierarchical IS-A relations such as "Nature → Energy → Energy conversion → Hydro-power"; and (ii) two synonym sources: MeSH alternative names and Wikipedia Synonyms[2] holding equivalent terms like "myocardial infarction" to "heart attack". For join tasks, (i) MED dataset[3] maps 293,294 research paper keywords to MeSH taxonomy node; and (ii) WIKI dataset[4] maps each of 1,212,943 Wikipedia articles to some categories. All algorithms were executed by JVM 8 on a Ubuntu computer with a Xeon 2.53GHz CPU and 32GB RAM.

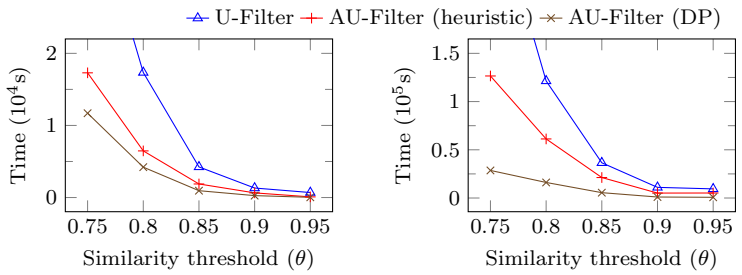
Join time. Figure 6.3 illustrates the time cost of joining two datasets. U-Filter is the normal prefix filtering presented in Section 6.2, AU-Filter (heuristic) and AU-Filter (DP) are multi-overlap filtering principles discussed in Subsection 6.2.1. From the figure, it is clear that both AU-Filter (heuristic) and AU-Filter (DP) outperform the baseline U-Filter, thanks to the adaptive algorithm which determine τ to enhance the filtering power. Furthermore, AU-Filter (DP) is the clear winner among all methods – five times faster than the AU-Filter (heuristic). This

[1] https://www.nlm.nih.gov/mesh/

[2] https://en.wikipedia.org/wiki/Wikipedia:LCM

[3] https://trec.nist.gov/data/t9_filtering.html

[4] https://wiki.dbpedia.org/



(a) MED (b) WIKI

Figure 6.3: Join time of proposed algorithms.

improvement owes to the dynamic programming that ensures short prefixes such that the number of false positives is reduced.

Filtering power. To further investigate the filtering power of each algorithm with various overlap constraints, we depicted the average length of pebble signatures and the number of candidates for different algorithms in Figure 6.4. It shows that AU-Filter (heuristic) can filter out 50% to 60% candidate pairs, while AU-Filter (DP) can prune away 70% to 90% pairs on both datasets. The dramatic reduction brought by AU-Filter can undoubtedly accelerate the verification process.

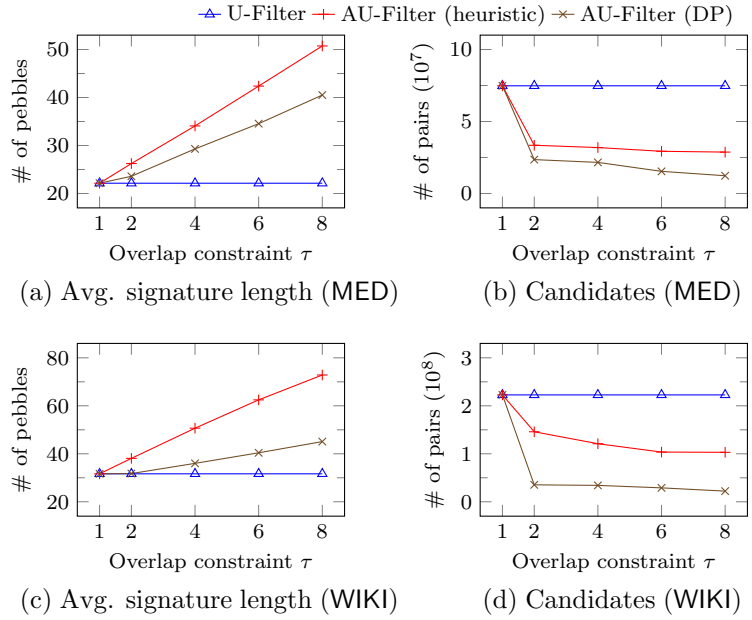
Scalability. To precisely evaluate the scalability of proposed filters, Figure 6.5 broke the join time into (i) the time to suggest the best parameter τ, (ii) filtering time, and (iii) verification time. It is clear that, by using our filtering algorithms, both filtering and verification times grow linearly instead of quadratically. Meanwhile, the cost of suggestion remains stable regardless of how the dataset changes, to about 15s for MED and 20s for WIKI datasets.

Parameter suggestion. To study the effect of the parameter τ on the overall join time, we implemented experiments to compare three different settings: (i) our suggested parameter, (ii) a random parameter, and (iii) the worst parameter. Table 6.5 compares the running time with various parameter settings for two datasets. As shown in the figure, our suggested parameter can achieve the best running time, which is far better than choosing the parameter randomly.

Comparison with existing algorithms. We finally compared our algorithm with four alternatives: pkduck[5] [44] for synonym similarity, K-Join [38] for taxonomy similarity, AdaptJoin[6] [48] for gram-based similarity, and their combination.

[5] `https://github.com/tracyhenry/xClean`

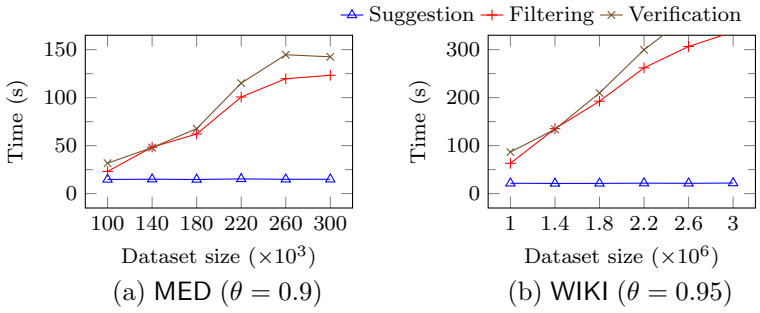
[6] `https://www.cs.sfu.ca/~jnwang/projects/adapt/`



(a) Avg. signature length (MED) (b) Candidates (MED)

(c) Avg. signature length (WIKI) (d) Candidates (WIKI)

Figure 6.4: Filtering power of various filters, $\theta = 0.85$.



(a) MED ($\theta = 0.9$) (b) WIKI ($\theta = 0.95$)

Figure 6.5: Scalability of AU-Filter (DP).

Table 6.5: Running time of AU-Filter (heuristic) w.r.t. different parameter selection methods.

Similarity threshold (θ)		0.75	0.80	0.85	0.90	0.95
MED (10^3s)	Using suggested τ	17.30	6.47	1.88	0.64	0.09
	Mean of random τ	24.81	9.47	2.70	0.92	0.27
	Using worst τ	45.04	17.94	4.39	1.36	0.72
WIKI (10^4s)	Using suggested τ	12.66	6.13	2.12	0.51	0.51
	Mean of random τ	22.55	8.30	2.85	0.70	0.69
	Using worst τ	34.41	13.01	3.91	1.18	0.89

Table 6.6: Effectiveness of our measure vs existing algorithms. P, R, and F stands for precision, recall, and F-measure, respectively.

Measure	MED, θ: 0.7			MED, 0.75			WIKI, 0.7			WIKI, 0.75		
	P	R	F	P	R	F	P	R	F	P	R	F
K-Join	0.89	0.12	0.20	0.86	0.09	0.17	0.83	0.08	0.15	0.83	0.05	0.10
AdaptJoin	0.81	0.19	0.30	0.79	0.15	0.25	0.71	0.28	0.40	0.64	0.15	0.24
pkduck	0.78	0.19	0.31	0.80	0.17	0.28	0.64	0.10	0.18	0.67	0.06	0.10
Combination	0.82	0.48	0.61	0.80	0.41	0.54	0.75	0.37	0.50	0.75	0.22	0.34
Ours	0.86	0.96	**0.91**	0.88	0.75	**0.81**	0.83	0.98	**0.90**	0.82	0.58	**0.68**

As shown in Table 6.6, our unified similarity measure achieved higher F-measures than each individual alternative as well as all of them together. Meanwhile, according to the join time summarised in Table 6.7, our algorithm consumed less time but found more similar strings than three alternatives combined, illustrating the superiority of our unified similarity measure as well as the adaptive filtering algorithm.

6.5 Chapter summary

This chapter introduced Paper IV, in which we proposed a unified filtering and verification framework to solve RQ4: finding similar strings from a mixture of typographical, synonymical, and taxonomical relevant words. Key contributions in this paper include: a new unified similarity measure as well as its approximation algorithm; "pebble", a structure that unifies various similarities; and two adaptive filtering principles, one heuristic and one dynamic programming, focusing on either speed or optimality.

Table 6.7: Join time (seconds) of our algorithm vs existing methods.

Method	MED (100K)					WIKI (100K)				
	θ: 0.75	0.8	0.85	0.9	0.95	0.75	0.8	0.85	0.9	0.95
K-Join	2.8	2.7	2.2	2.1	1.8	5.4	5.0	3.1	3.0	2.7
Ours (only taxonomy)	2.6	2.6	2.2	2.1	1.8	4.5	4.3	2.9	2.9	2.7
AdaptJoin	1045.8	675.4	270.6	48.3	10.5	1360.2	1044.6	480.6	120.0	16.9
Ours (only Jaccard)	597.9	217.0	85.5	20.9	10.3	1301.3	644.5	274.2	62.6	11.3
pkduck	51.6	30.6	15.1	8.3	7.4	39.5	17.3	8.6	3.5	1.4
Ours (only synonym)	20.8	18.0	14.7	7.0	6.8	15.8	11.1	8.0	4.3	3.2
Combination	1100.2	708.7	287.9	58.7	19.1	1405.1	1066.9	492.3	126.5	21.0
Ours (all measures)	842.1	413.8	253.7	54.7	18.9	1418.1	694.7	308.9	113.5	22.4

Chapter 7

Conclusions and future work

In this chapter, we conclude this thesis by revisiting Chapters 3 to 6. Furthermore, we present some interesting research directions that might be worth investigating further.

7.1 Conclusions

Strings are one of the most common forms of data which play a crucial role in numerous applications. Since a string can contain inconsistencies, such as typographical errors and representational variations, many pieces of research are conducted on finding similar strings in an approximate way.

In this thesis, we solved a group of semantic approximate string matching problems, which is to find relevant strings considering synonyms and taxonomies. We formalised four research problems and proposed algorithms to solve each problem efficiently, as well as performed theoretical analysis to prove their correctness.

We first studied the top-k selection problem by considering synonyms. Three algorithms are proposed here, two of which use sorted lists and one uses tries, each has a specific optimisation goal: Twin Tries aims to minimise space cost by using two tries, Expansion Trie is to improve the lookup performance by attaching synonyms to the string itself, and Hybrid Trie to strike a balance between space cost and selection time. We showed the NP-hardness of constructing a Hybrid Trie, for which we proposed a new branch and bound-based solution. Experimental results showed that the selection speed of the Hybrid Trie is on a par with that of Expansion Trie while it consumes far less space than Twin Tries.

We then investigated the problem of finding the top-k similar strings w.r.t. taxonomy knowledge. To this end, we proposed (i) two list-based algorithms, in which we developed pre-computed LP values to avoid LCA computations thus accelerating the join process; and (ii) one trie-based algorithm, which makes use of the trie structure to eliminate unnecessary accesses to unfeasible strings. Experimental results showed that our optimisations significantly improved the top-k join performances, and particularly, the trie-based algorithm was able to maintain high performance on a huge dataset.

Next, we considered the problem of performing all-match joins given taxonomy knowledge. Here, we identified the problem of the standard prefix filtering, where the number of false positives is significant due to the loose condition for two strings being potentially similar. Notable contributions include a new multi-overlap prefix filtering principle which significantly reduced the number of false positives after the filtering stage, as well as an estimation algorithm that makes use of multiple samples to find the best overlap parameter in a short time.

We finally researched an all-match join problem by integrating multiple inconsistencies, namely typographical errors, synonyms, and taxonomically-related words. We proposed a new unified similarity measure that incorporates multiple measures to find the overall similarity of two strings. We proved that finding the maximal string similarity is NP-hard, and subsequently designed a non-trivial approximation algorithm that runs in polynomial time and has a tight error bound. As for actual join tasks, we proposed two prefix filtering principles, backed by a fast heuristic and an accurate dynamic programming protocol, respectively. Extensive experiments showed the superiority of our algorithm over the state of the art, which motives for its use in practice.

7.2 Future work

As the first direction, it would be interesting to investigate a way to integrate stemming [39, 40], a process in the information retrieval (IR) field that deals with grammatical variations, into the approximate string search/join process. Such integration is expected to increase the effectiveness of proposed algorithms since when the input dataset is not stemmed beforehand, our proposed algorithms would be insufficient to obtain high similarities. For example, the 2-gram Jaccard similarity of "ox" and "oxen" is only $\frac{1}{3}$, but they, in most cases, should be treated as the same word. One straightforward solution to this problem is to use synonyms, e.g. "ox $\rightarrow$ oxen", but a more elegant solution is undoubtedly desirable.

The second direction is to consider *context* when evaluating strings' similarities. As an example, since "renting" and "leasing" are synonyms, the similarity between the two strings "renting an apartment" and "leasing an apartment" is high according to our algorithms – which is correct. However, if the strings are "renting a car" and "leasing a car", the high similarity returned by our algorithms leads to a false positive, because their meanings are different[1]. It would be nice if our proposed algorithms could choose whether to apply synonyms/taxonomies depending on the context.

Another research direction would be about extending the unified similarity measure discussed in Subsection 6.1.1. To this end, one interesting extension is *word vector* [29, 33]. It works by mapping words into numerical vectors of high dimensions so that the similarity of two words is equivalent to the distance of corresponding vectors. The biggest challenge here is that, since vector distance does not satisfy the triangle inequality [32, 34], we are currently not able to use word vector models directly as a string similarity metric.

Finally, since the string dataset is often huge and requires hours – even days – to process on a single machine, one practical and useful extension is to adapt the proposed algorithms for use in distributed environments. This idea is not new and has been researched over the last decade [13, 16, 36, 37, 62], but most of them focus on typographical similarity measures. When extending them to synonyms, taxonomies, or even the unified measures, it would be a challenge to design new algorithms for generating string signatures to avoid losing any true positive, as well as appropriate partition schemes to minimise the amount of data shuffling between workers.

[1] "car renting": `https://en.wikipedia.org/wiki/Car_rental` vs "car leasing": `https://en.wikipedia.org/wiki/Vehicle_leasing`

References

[1] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Transformation-based framework for record matching. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 40–49. IEEE Computer Society, 2008.

[2] Esther M. Arkin and Refael Hassin. On local search for weighted k-set packing. In Rainer E. Burkard and Gerhard J. Woeginger, editors, *Proceedings of the Algorithms - ESA '97, 5th Annual European Symposium, Graz, Austria, September 15-17, 1997, Proceedings*, volume 1284 of *Lecture Notes in Computer Science*, pages 13–22. Springer, 1997.

[3] Esther M. Arkin and Refael Hassin. On local search for weighted k-set packing. *Mathematics of Operations Research*, 23(3):640–648, 1998.

[4] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. Dbpedia: A nucleus for a web of open data. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007.*, pages 722–735, 2007.

[5] Vineet Bafna, Babu O. Narayanan, and R. Ravi. Nonoverlapping local alignments (weighted independent sets of axis-parallel rectangles). *Discrete Applied Mathematics*, 71(1-3):41–53, 1996.

[6] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 131–140. ACM, 2007.

[7] Piotr Berman. A $d/2$ approximation for maximum weight independent set in d-claw free graphs. In *Algorithm Theory - SWAT 2000, Proceedings of the 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000*, pages 214–219, 2000.

[8] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. Spatio-textual similarity joins. *Proceedings of the Very Large Databases Endowment*, 6(1):1–12, 2012.

[9] Jennifer J. Burg, John D. Ainsworth, Brian Casto, and Sheau-Dong Lang. Experiments with the "oregon trail knapsack problem". *Electronic Notes in Discrete Mathematics*, 1:26–35, 1999.

[10] Barun Chandra and Magnús M. Halldórsson. Greedy local improvement and weighted set packing approximation. *Journal of Algorithms*, 39(2):223–240, 2001.

[11] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 5. IEEE Computer Society, 2006.

[12] Surajit Chaudhuri and Raghav Kaushik. Extending autocompletion to tolerate errors. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 707–718. ACM, 2009.

[13] Zhimin Chen, Yue Wang, Vivek R. Narasayya, and Surajit Chaudhuri. Customizable and scalable fuzzy join for big data. *Proceedings of the Very Large Databases Endowment*, 12(12):2106–2117, 2019.

[14] Francesco Concas, Pengfei Xu, Mohammad A Hoque, Jiaheng Lu, and Sasu Tarkoma. Multiple set matching with bloom matrix and bloom vector. *ACM Transactions on Knowledge Discovery from Data*, 14(2):1–21, 2020.

[15] George B Dantzig. Discrete-variable extremum problems. *Operations research*, 5(2):266–288, 1957.

[16] Dong Deng, Guoliang Li, Shuang Hao, Jiannan Wang, and Jianhua Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In

Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski, editors, *Proceedings of the 30th IEEE International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 340–351. IEEE Computer Society, 2014.

[17] Michael Färber, Frederic Bartscherer, Carsten Menne, and Achim Rettinger. Linked data quality of dbpedia, freebase, opencyc, wikidata, and YAGO. *Semantic Web*, 9(1):77–129, 2018.

[18] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 491–500. Morgan Kaufmann, 2001.

[19] Marios Hadjieleftheriou and Divesh Srivastava. Approximate string matching. *Foundations and Trends in Databases*, 2(4):267–402, 2011.

[20] Cor A. J. Hurkens and Alexander Schrijver. On the size of systems of sets every t of which have an sdr, with an application to the worst-case ratio of heuristics for packing problems. *SIAM Journal on Discrete Mathematics*, 2(1):68–72, 1989.

[21] Shengyue Ji, Guoliang Li, Chen Li, and Jianhua Feng. Efficient interactive fuzzy keyword search. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl, editors, *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 371–380. ACM, 2009.

[22] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.

[23] Peter J Kolesar. A branch and bound algorithm for the knapsack problem. *Management science*, 13(9):723–735, 1967.

[24] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 257–266. IEEE Computer Society, 2008.

[25] Chen Li, Bin Wang, and Xiaochun Yang. VGRAM: improving performance of approximate queries on string collections using variable-length grams. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 303–314. ACM, 2007.

[26] Jiaheng Lu, Chunbin Lin, Wei Wang, Chen Li, and Haiyong Wang. String similarity measures and joins with synonyms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 373–384, 2013.

[27] Jiaheng Lu, Chunbin Lin, Wei Wang, Chen Li, and Xiaokui Xiao. Boosting the quality of approximate string matching by synonyms. *ACM Transactions on Database Systems*, 40(3):15:1–15:42, 2015.

[28] Sean P. Meyn and Richard L. Tweedie. *Markov Chains and Stochastic Stability*. Communications and Control Engineering Series. Springer, 1993.

[29] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *Workshop Track Proceedings of the 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA*, 2013.

[30] Prasenjit Mitra. Dewey decimal system. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 808–809. Springer US, 2009.

[31] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38, 1957.

[32] Aida Nematzadeh, Stephan C. Meylan, and Thomas L. Griffiths. Evaluating vector-space models of word representation, or, the unreasonable effectiveness of counting words near other words. In Glenn Gunzelmann, Andrew Howes, Thora Tenbrink, and Eddy J. Davelaar, editors, *Proceedings of the 39th Annual Meeting of the Cognitive Science Society, CogSci 2017, London, UK, 16-29 July 2017*. cognitivesciencesociety.org, 2017.

[33] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL, 2014.

[34] Joseph Reisinger and Raymond J. Mooney. Multi-prototype vector-space models of word meaning. In *Human Language Technologies: Proceedings of the Conference of the North American Chapter of the Association of Computational Linguistics, June 2-4, 2010, Los Angeles, California, USA*, pages 109–117. The Association for Computational Linguistics, 2010.

[35] Leonardo Andrade Ribeiro and Theo Härder. Generalizing prefix filtering to improve set similarity joins. *Information Systems*, 36(1):62–78, 2011.

[36] Chuitian Rong, Chunbin Lin, Yasin N. Silva, Jianguo Wang, Wei Lu, and Xiaoyong Du. Fast and scalable distributed set similarity joins for big data analytics. In *Proceedings of the 33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1059–1070. IEEE Computer Society, 2017.

[37] Chuitian Rong, Wei Lu, Xiaoli Wang, Xiaoyong Du, Yueguo Chen, and Anthony K. H. Tung. Efficient and scalable processing of string similarity join. *IEEE Transactions on Knowledge and Data Engineering*, 25(10):2217–2230, 2013.

[38] Zeyuan Shang, Yaxiao Liu, Guoliang Li, and Jianhua Feng. K-join: Knowledge-aware similarity join. *IEEE Transactions on Knowledge and Data Engineering*, 28(12):3293–3308, 2016.

[39] Jasmeet Singh and Vishal Gupta. Text stemming: Approaches, applications, and challenges. *ACM Computing Surveys*, 49(3):45:1–45:46, 2016.

[40] Jasmeet Singh and Vishal Gupta. A systematic review of text stemming techniques. *Artificial Intelligence Review*, 48(2):157–217, 2017.

[41] Kristian Slabbekoorn, Laura Hollink, and Geert-Jan Houben. Domain-aware ontology matching. In *The Semantic Web - ISWC 2012 - Proceedings of the 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Part I*, pages 542–558, 2012.

[42] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 697–706, 2007.

[43] Thomas Pellissier Tanon, Denny Vrandecic, Sebastian Schaffert, Thomas Steiner, and Lydia Pintscher. From freebase to wikidata: The great migration. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 1419–1428, 2016.

[44] Wenbo Tao, Dong Deng, and Michael Stonebraker. Approximate string joins with abbreviations. *Proceedings of the Very Large Databases Endowment*, 11(1):53–65, 2017.

[45] David Vengerov, Andre Cavalheiro Menck, Mohamed Zaït, and Sunil Chakkappen. Join size estimation subject to filter conditions. *Proceedings of the Very Large Databases Endowment*, 8(12):1530–1541, 2015.

[46] Hongya Wang, Lihong Yang, and Yingyuan Xiao. Setjoin: a novel top-k similarity join algorithm. *Soft Computing*, pages 1–16, 2020.

[47] Jiannan Wang, Guoliang Li, and Jianhua Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *Proceedings of the Very Large Databases Endowment*, 3(1):1219–1230, 2010.

[48] Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 85–96. ACM, 2012.

[49] Jiaying Wang, Xiaochun Yang, Bin Wang, and Chengfei Liu. Ls-join: Local similarity join on string collections. *IEEE Transactions on Knowledge and Data Engineering*, 29(9):1928–1942, 2017.

[50] Pei Wang, Chuan Xiao, Jianbin Qin, Wei Wang, Xiaoyang Zhang, and Yoshiharu Ishikawa. Local similarity search for unstructured text. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference*

2016, San Francisco, CA, USA, June 26 - July 01, 2016, pages 1991–2005. ACM, 2016.

[51] Zhibiao Wu and Martha Stone Palmer. Verb semantics and lexical selection. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics, 27-30 June 1994, New Mexico State University, Las Cruces, New Mexico, USA*, pages 133–138, 1994.

[52] Cao Xiao, David Mandell Freeman, and Theodore Hwa. Detecting clusters of fake accounts in online social networks. In Indrajit Ray, Xiaofeng Wang, Kui Ren, Christos Dimitrakakis, Aikaterini Mitrokotsa, and Arunesh Sinha, editors, *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security, AISec 2015, Denver, Colorado, USA, October 16, 2015*, pages 91–101. ACM, 2015.

[53] Chuan Xiao, Jianbin Qin, Wei Wang, Yoshiharu Ishikawa, Koji Tsuda, and Kunihiko Sadakane. Efficient error-tolerant query autocompletion. *Proceedings of the Very Large Databases Endowment*, 6(6):373–384, 2013.

[54] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of the Very Large Databases Endowment*, 1(1):933–944, 2008.

[55] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. Top-k set similarity joins. In Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors, *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 916–927. IEEE Computer Society, 2009.

[56] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors, *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 131–140. ACM, 2008.

[57] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems*, 36(3):15:1–15:41, 2011.

[58] Pengfei Xu and Jiaheng Lu. Top-k string auto-completion with synonyms. In *Database Systems for Advanced Applications - Proceedings of the 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Part II*, pages 202–218, 2017.

[59] Pengfei Xu and Jiaheng Lu. Efficient taxonomic similarity joins with adaptive overlap constraint. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, pages 1563–1566, 2018.

[60] Pengfei Xu and Jiaheng Lu. Efficient string similarity join with taxonomy knowledge. *Submitted to Knowledge and Information Systems*, 2019.

[61] Pengfei Xu and Jiaheng Lu. Towards a unified framework for string similarity joins. *Proceedings of the Very Large Databases Endowment*, 12(11):1289–1302, 2019.

[62] Cairong Yan, Xue Zhao, Qinglong Zhang, and Yongfeng Huang. Efficient string similarity join in multi-core and distributed systems. *PloS one*, 12(3), 2017.

[63] Xiaochun Yang, Bin Wang, and Chen Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 353–364. ACM, 2008.

[64] Xiaochun Yang, Yaoshu Wang, Bin Wang, and Wei Wang. Local filtering: Improving the performance of approximate queries on string collections. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 377–392. ACM, 2015.

[65] Minghe Yu, Jin Wang, Guoliang Li, Yong Zhang, Dong Deng, and Jianhua Feng. A unified framework for string similarity search with edit-distance constraint. *The VLDB Journal*, 26(2):249–274, 2017.